

Integration, Indexierung und Interaktion hochdimensionaler Datenobjekte

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät Informatik
von

Konstantin Koll

Dortmund
2009

Tag der mündlichen Prüfung:	18. Februar 2009
Dekan:	Prof. Dr. Peter Buchholz
Gutachter:	Prof. Dr. Gisbert Dittrich Prof. Dr. Heinrich Müller

Ich bedanke mich bei Prof. Dr. Dittrich und Prof. Dr. Müller von der Technischen Universität Dortmund für die Betreuung meiner Promotion, ebenso bei Prof. Dr. Spinczyk für seine Unterstützung. Besonderer Dank gilt Judith Ahrens, Lars Heide, Diether Koll, Felix Kaiser, Felizitas Ottemeier, Ingo-Olaf Schumacher, Jindra Siekmann, Sandra Teusner und Vasilis Vasaitis für viele Verbesserungsvorschläge.

Inhaltsverzeichnis

Anhang

Begriffe aus dem Glossar sind bei der ersten Verwendung *kursiv* gedruckt.

1 Einleitung

Die Anforderungen an Computersysteme haben sich in den letzten Jahren besonders im privaten Bereich deutlich verändert, hin zu einem Speicher- und Wiedergabesystem für digitale Medien wie Musik, Videos und Fotos. Diese Entwicklung wird durch die noch immer wachsende Beliebtheit von MP3-Dateien und ihren Derivaten, digitaler Fotografie und neuerdings auch durch digitales hochauflösendes Fernsehen angetrieben. Sie hat ihren vorläufigen Höhepunkt in sog. »Mediacenter«-Applikationen gefunden, die digitale Medien endgültig im heimischen Wohnzimmer ankommen lassen **[Ker03]**.

Leider sind die von Betriebssystemen eingesetzten Dateisysteme nicht weiterentwickelt worden, um den veränderten Anforderungen beim Speichern und vor allem Wiederfinden tausender Dateien gerecht zu werden. Lediglich die physikalische Größe von Dateisystemen ist durch Einführung größerer Adressräume und effizienterer Datenstrukturen gewachsen. Auf logischer Ebene bieten die heute eingesetzten Dateisysteme als Ordnungsschema noch immer eine Verzeichnishierarchie an, die jedoch deutliche Nachteile aufweist. **[Dou00]** identifiziert einige dieser Nachteile, die teilweise bereits in **[Mal83]** beschrieben worden sind, dort allerdings bezogen auf Papierakten. Zunächst können Dateien nur an genau einem Platz in der Verzeichnishierarchie abgelegt werden. Ein Beispiel hierfür ist der folgende Verzeichnisbaum, in den ein fiktiver Nutzer seine Fotos einsortiert:

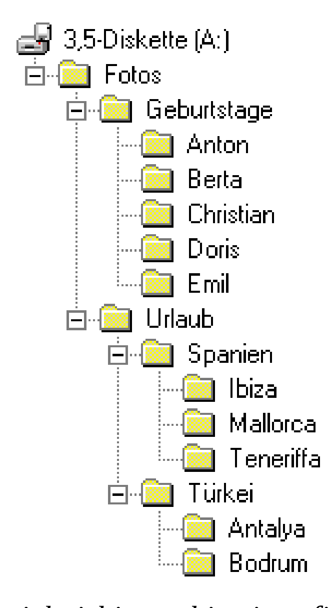


Abb. 1-1: Verzeichnishierarchie eines fiktiven Nutzers

Auf der obersten Ebene sind alle Fotos in Aufnahmen von Geburtstagsfeiern und Urlaubsfotos eingeteilt – erstere dann nach Personen, letztere nach Ländern und schließlich weiter nach Orten. Dieses System weist jedoch prinzipielle Lücken auf: Wo sollen beispielsweise Fotos von Bertas Geburtstag abgelegt werden, wenn sie ihn auf Mallorca gefeiert hat ? Wenn entsprechende Bilder in beiden Zusammenhängen auffindbar sein sollen, müssen sie auch über beide Ordner zugänglich gemacht werden, etwa mittels Einführung symbolischer Links. Das Anlegen weiterer Kategorien, etwa des Aufnahmejahres, steigert diesen Aufwand zusehends weiter.

Darüber hinaus ist eine solche Verzeichnishierarchie auf die Mitarbeit des Benutzers angewiesen **[Mil05]**, der geeignete Kategorien für seine Dateien finden, entsprechende Unterverzeichnisse anlegen und dann auch einsetzen muss. In der Praxis wird es zudem oft vorkommen, dass Dateien in den falschen Verzeichnissen gespeichert werden, so dass sie nicht mehr aufgefunden werden können und »verloren gehen«.

Für diesen Fall bieten Betriebssysteme mehr oder weniger gute Suchfunktionen an, die als Suchkriterium jedoch nur die Attribute erlauben, die das Dateisystem selbst verwaltet. Weitere Metadaten, die insbesondere in modernen Multimedia-Dateiformaten enthalten sind, stehen oft nur in einigen Programmen zur Verfügung, die solche Dateiformate öffnen können. Auf Betriebssystem-Ebene sind diese Attribute unzugänglich, obwohl gerade sie eine sinnvolle Suche ermöglichen würden.

1.1 Zielsetzung und Überblick

Das Hauptziel dieser Arbeit besteht darin, den Zugriff von Benutzern auf ihre Dateien, insbesondere auf Multimedia-Inhalte, zu verbessern. Diese Aufgabenstellung erfordert die interdisziplinäre Bearbeitung der Bereiche Integration (→ *Kap. 1.1.1*), Indexierung (→ *Kap. 1.1.2*) und Interaktion (→ *Kap. 1.1.3*). Auf der Basis einer Referenz-Library, welche die Integration und Indexierung von hochdimensionalen Datenobjekten realisiert, kann die Interaktion durch diverse Verbesserungen optimiert werden:

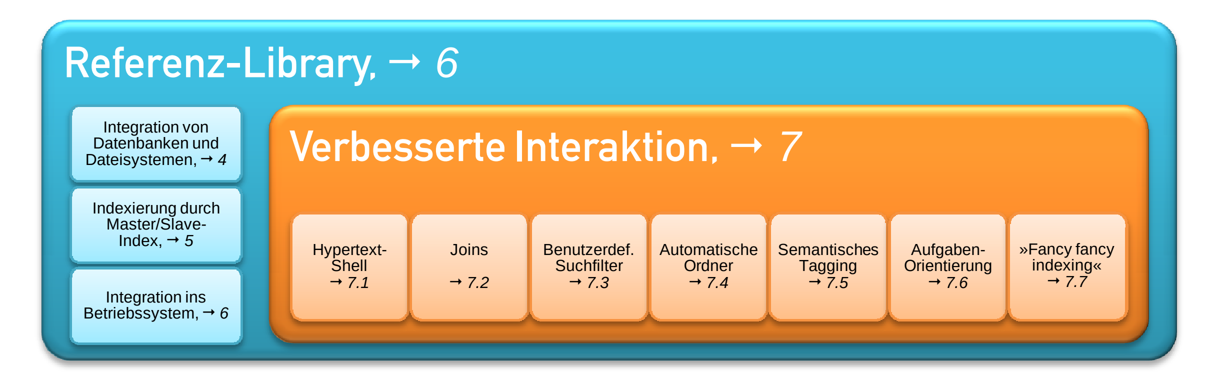


Abb. 1.1-1: Resultate (blau) und ihre Vorteile für den Benutzer (orange)

Im Zusammenspiel erfüllen diese Innovationen das oben gesetzte Ziel eines verbesserten Zugriffs auf eine große Anzahl Dateien. Abschließend werden Möglichkeiten zur Integration in bereits existierende Systeme sowie zur Weiterentwicklung vorgestellt (→ *Kap. 8*).

1.1.1 Integration

Als Teilziel »Integration« wird der uneingeschränkte Zugriff auf alle Attribute von Dateien und Tupeln relationaler Datenbanken auf Betriebssystem-Ebene realisiert.

Marsden stellt hierzu in **[Mar03]** fest, dass Datenbanken ursprünglich in hierarchielosen Dateien gespeichert wurden. Später wurden hierarchische Datenbanken eingeführt, die die Darstellung von 1:*n*-Beziehungen zwischen Entitäten erlaubten. Durch ein hierarchisches Datenmodell konnten jedoch nicht alle Beziehungen zwischen Entitäten modelliert werden, weshalb Links eingeführt wurden, um die Hierarchie zu durchbrechen. Auf dieser Entwicklungsstufe befinden sich auch herkömmliche Dateisysteme **[Mar03]**. Inzwischen hat sich in der Datenbanktechnik jedoch das relationale Datenmodell **[Cod70]** durchgesetzt **[Saa05]**, mit dem Beziehungen dynamisch durch die Daten selbst modelliert werden.

Gifford et al. haben in **[Gif91]** den Begriff des *semantischen Dateisystems* eingeführt. Damit sind Dateisysteme gemeint, die beliebige Attribute (Metadaten) zu Dateien abspeichern und verwalten können. In → *Kap. 4* wird das Datenmodell einer »Library« eingeführt, auf das sich die Modelle von Dateisystemen und relationalen Datenbanken abbilden lassen. Auf diese Weise werden beide Systeme auf logischer Ebene vereinigt, so dass alle Attribute für Applikationen zugänglich sind **[Sho93]** und globale Gültigkeit **[Imf02]** erlangen. Beziehungen werden durch die Attribute selbst modelliert, so dass keine von außen erzwungenen Hierarchien wie Verzeichnisbäume mehr benötigt werden.

1.1.2 Indexierung

Die Attribute von Datenobjekten müssen indexiert werden, um auch bei großen Datenmengen einen ausreichend schnellen Zugriff zu erzielen. Speziell bei der Indexierung von Dateiattributen werden Probleme durch den hochdimensionalen Datenraum verursacht (»Fluch der Dimensionen«, → *Kap. 3.3.1*). Ebenfalls negativ wirkt sich aus, dass der Index in der Regel innerhalb eines physikalischen Dateisystems gespeichert wird (→ *Kap. 3.3.3*).

Das Teilziel »Indexierung« besteht darin, die oben genannten Probleme beim Indexieren von Dateiattributen zu lösen. Dies wird durch eine neuartige Datenstruktur erreicht, die anhand einer Referenz-Implementierung erprobt wird. Der »Master/Slave-Index« (→ *Kap. 5*) ist für den Einsatz in hochdimensionalen Datenräumen, die von heterogenen Objekt-Schemata erzeugt werden, geeignet. Da außerdem alle Algorithmen sequenziell arbeiten und ohne Neupositionierung des Dateizeigers auskommen, wird durch eine zusätzliche Komprimierung des Indexes eine weitere Leistungssteigerung erzielt.

1.1.3 Interaktion

Basierend auf der Integration und Indexierung wird als letztes Teilziel die Interaktion mit großen Datenmengen durch die Realisierung fortschrittlicher Benutzerschnittstellen verbessert. In einer Referenz-Library können sehr viele Dateien anhand ihrer Metadaten sowie weiterer, vom Benutzer vergebener Attribute schnell organisiert und aufgefunden werden.

1.2 Resultate

Im Rahmen dieser Arbeit wurden folgende Resultate mit Neuheitswert erzielt:

- Traditionell werden die Gebiete »Betriebssysteme« sowie »Datenbanken« als unabhängige Disziplinen angesehen. Daher bestand ein Bedarf an einer umfassenden Bestandsaufnahme bereits existierender Ansätze zur Verbesserung von Dateisystemen (→ *Kap. 2*). Im Zuge dieser Bestandsaufnahme wurden auch eingesetzte Indexe untersucht (→ *Kap. 3*).
- Das Datenmodell für Libraries, auf das sich die Modelle von relationalen Datenbanken und Dateisystemen abbilden lassen, wird eingeführt (→ *Kap. 4*).
- Der »Master/Slave-Index« wird als neue Datenstruktur zur Indexierung eingeführt (→ *Kap. 5*).
- Zum Nachweis der praktischen Funktionsfähigkeit des Library-Modells wird eine Referenz-Library vorgestellt (→ *Kap. 6*).
- Auf Basis einer Library als schnelles Speichersystem für Datenobjekte mit beliebigen Attributen werden Verbesserungen an Benutzungsschnittstellen, und damit bei der Interaktion mit Dateien, beschrieben (→ *Kap. 7*).

1.3 Veröffentlichungen

Die Resultate dieser Arbeit (→ *Kap. 1.2*) stützen sich auf diverse Veröffentlichungen, deren alleiniger Autor Konstantin Koll ist. In **[Kol08a]** wird die Benutzung möglichst großer Zuordnungseinheiten in Dateisystemen zur Steigerung der Zugriffsgeschwindigkeit gefordert. **[Kol08b]** beschreibt die Architektur der Referenz-Library sowie die Schwierigkeiten bei ihrer Implementierung, insbesondere bezüglich der Indexierung (→ *Kap. 6*). Der Master/Slave-Index (→ *Kap. 5*) und seine Leistungsfähigkeit (→ *Kap. 6.3*) wird ausführlich in **[Kol08c]** beschrieben. Zuletzt waren Verbesserungen an Webserver-Applikationen (→ *Kap. 7.7*) Gegenstand eines Vortrages **[Kol08d]**.

Darüber hinaus wurde 2007 der Master/Slave-Index (→ *Kap. 5*) beim U.S. Patentamt unter der Nummer 11/892071 unter Berufung auf ein provisorisches Patent vom 18.09.2006 zum Patent angemeldet **[Kol07a]**.

2 Existierende Lösungen

Seit langer Zeit wird an Möglichkeiten gearbeitet, die Organisation von Dateisystemen zu optimieren. In den letzten Jahren erfreuen sich vor allem sog. »Desktop-Suchmaschinen« (→ *Kap. 2.10*) großer Beliebtheit, die von Suchmaschinen-Betreibern wie Google und Yahoo für lokale Dateisysteme entwickelt wurden; ähnliche Programme existieren schon seit Anfang der 1990er-Jahre (Lotus Magellan, → *Kap. 2.1*).

In diesem Kapitel werden verschiedene Ansätze untersucht, dokumentiert und bewertet (→ *Kap. 2.11*). Neben Desktop-Suchmaschinen werden auch relevante Entwicklungen aus dem akademischen Bereich und Techniken, die lediglich Teilaufgaben lösen, **chronologisch** vorgestellt. Viele Ideen dieser Lösungen und Prototypen sind dabei in Weiterentwicklungen eingeflossen. Die untersuchten Lösungen sind sehr unterschiedlicher Natur, sowohl hinsichtlich ihrer Zielsetzung als auch ihrer Arbeitsweise. Dadurch wird ein direkter Vergleich erschwert.

Es lassen sich jedoch zwei unterschiedliche Ansätze definieren, die in einigen Systemen sogar kombiniert werden. Zum einen wird versucht, Dateien und Applikationen in eine objektorientierte Gesamtarchitektur einzubinden. Diese Entwicklung hat mit dem Rufus-Projekt (→ *Kap. 2.3*) und Microsoft OLE (→ *Kap. 2.4*) begonnen, und wurde mit moderneren Systemen wie Microsoft WinFS (→ *Kap. 2.9*) fortgesetzt. Andererseits wird versucht, die eingangs erwähnten Schwächen physikalischer Dateisysteme (→ *Kap. 1*) durch Weiterentwicklungen oder Zusatzsoftware zu beheben. Das Rufus-Projekt, BeFS (→ *Kap. 2.7*) und Microsoft WinFS lassen sich beiden Kategorien zuordnen.

2.1 Lotus Magellan

1990 wurde von der Firma Lotus das DOS-Programm »Magellan« als eine der ersten Desktop-Suchmaschinen veröffentlicht. Moderne Desktop-Suchmaschinen (→ *Kap. 2.10*) basieren noch immer auf dem Architekturmodell, das mit Magellan eingeführt wurde:

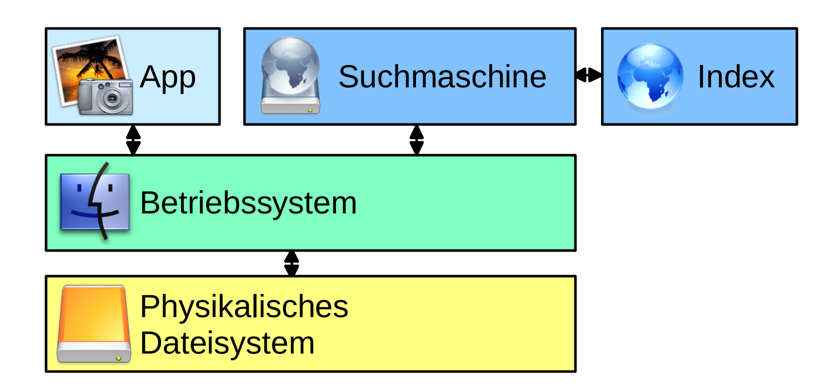


Abb. 2.1-1: Architektur einer Desktop-Suchmaschine

Desktop-Suchmaschinen sind normale Applikationen, die über das Betriebssystem auf das Dateisystem zugreifen. Die Suchmaschine wird erst aktiv, wenn der Anwender nach Dateien sucht, die bestimmte Metadaten oder Schlüsselworte enthalten.

1990 existierten viele der heute verbreiteten Dateiformate wie PDF, MP3, JPEG, QuickTime oder AVI **[Kol03] [Kol07b]** noch nicht, so dass auch die Metadaten-Attribute dieser Dateitypen weitgehend unbekannt waren. Aus diesem Grund unterstützt Magellan lediglich das Durchsuchen von Dateien nach Schlüsselworten und verwendet einen Volltext-Index. Die Extraktion von Textstellen wird dabei von Plug-Ins (»Viewer« genannt) übernommen. Im Lieferumfang befinden sich Viewer für viele damals benutzte Dateiformate, unter anderem:

- Ascii-Text
- CompuServe
- dBase
- GIF
- Lotus 1-2-3
- Lotus Agenda
- Lotus Manuscript
- Lotus Symphony
- LotusWorks
- Microsoft Excel
- Microsoft Word
- Multiplan
- Paradox
- PCX
- Quark Express
- Quicken
- Quattro Pro
- WordPerfect
- Wordstar

Abb. 2.1-2: von Lotus Magellan unterstützte Dateiformate

Da die Magellan-Suchmaschine aufgrund ihres Alters besonders einfach aufgebaut ist, kann sie hier exemplarisch die Funktionsweise moderner Desktop-Suchmaschinen (→ *Kap. 2.10*) veranschaulichen, die ausnahmslos nach demselben Prinzip arbeiten.

2.1.1 Arbeitsweise

Nach dem Programmstart erscheint die Magellan-Oberfläche, die in der linken Spalte alle Dateien von allen Datenträgern des Computers enthält. Die Verzeichnisstruktur wird dabei vollständig ignoriert, denn Magellan dient ja gerade dem Auffinden von Dateien mit unbekanntem Pfad:

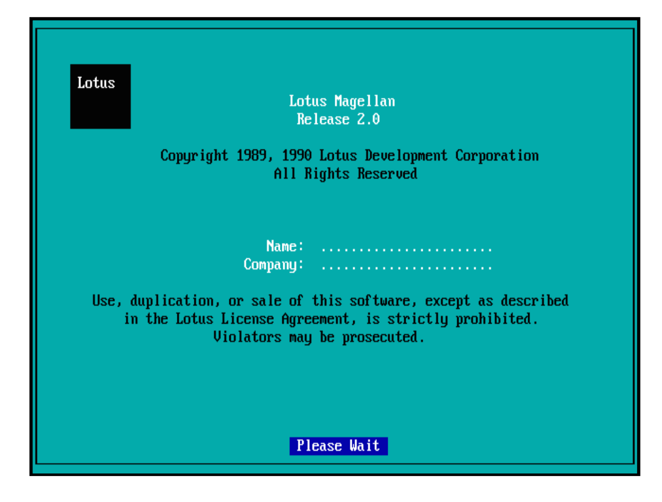


Abb. 2.1-3: Programmstart

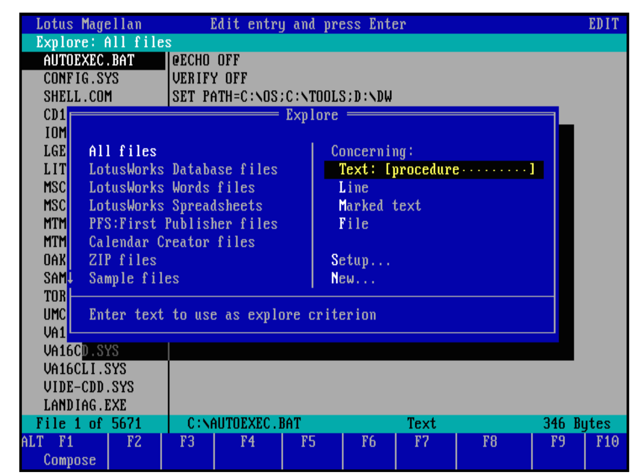


Abb. 2.1-4: Suchen nach einem Begriff

Beim ersten Programmstart kann die Suchanfrage allerdings noch nicht bearbeitet werden, da Magellan noch keine Dateien indexiert hat:

```
Magellan could not open the index file with
the .IX2 extension.

▪ If the index resides on a network, another
  user may be updating or rebuilding it.
▪ Use Index (Alt-F5) Update to update the index
  and try the search again.
▪ Consider the possibility of a disk error.

Press Esc to continue.
```

Abb. 2.1-5: Fehlender Index

Vor dem Indexieren können bestimmte Dateiendungen angegeben werden. *:* beschreibt dabei alle Dateien auf allen logischen Laufwerken, ausgenommen werden jedoch Dateien mit den Extensionen, denen ein Minuszeichen vorangestellt ist (hier also alle ausführbaren Dateien):

```
Index Setup

Description:     [All files················]
Index file name: [ALLFILES························]
Filespec:        [*:* -*.exe -*.com -*.sys -*.cal·]

Enter or edit descriptive name of index (type ^ to
highlight the character that follows it)
```

Abb. 2.1-6: Zu indexierende Dateitypen

Im Anschluss werden die oben ausgewählten Dateien ohne weitere Eigenaktivität des Benutzers in ca. 40 Minuten indexiert:

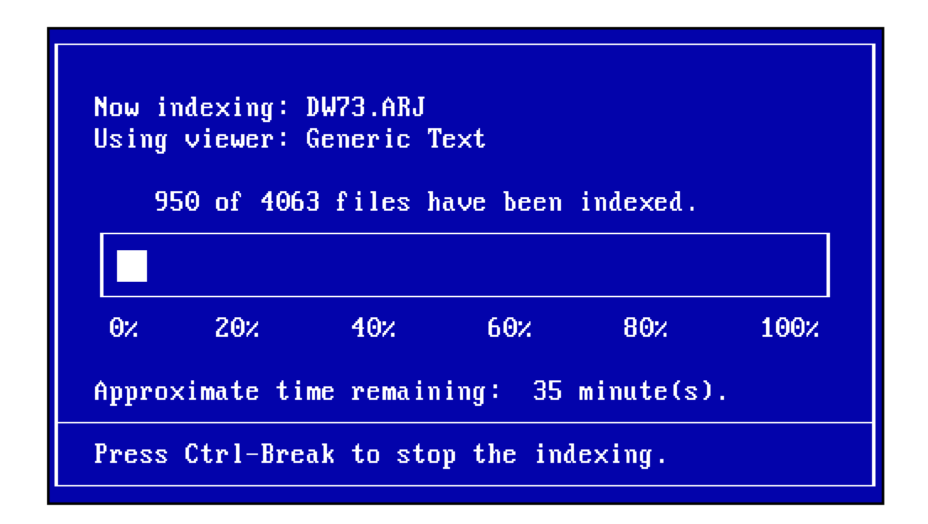


Abb. 2.1-7: Indexierung

Nach dem Indexieren der Dateien ist die Suche nach dem Begriff »procedure« erfolgreich und liefert eine Dateiliste, die hier fast ausschließlich aus Quelltexten besteht:

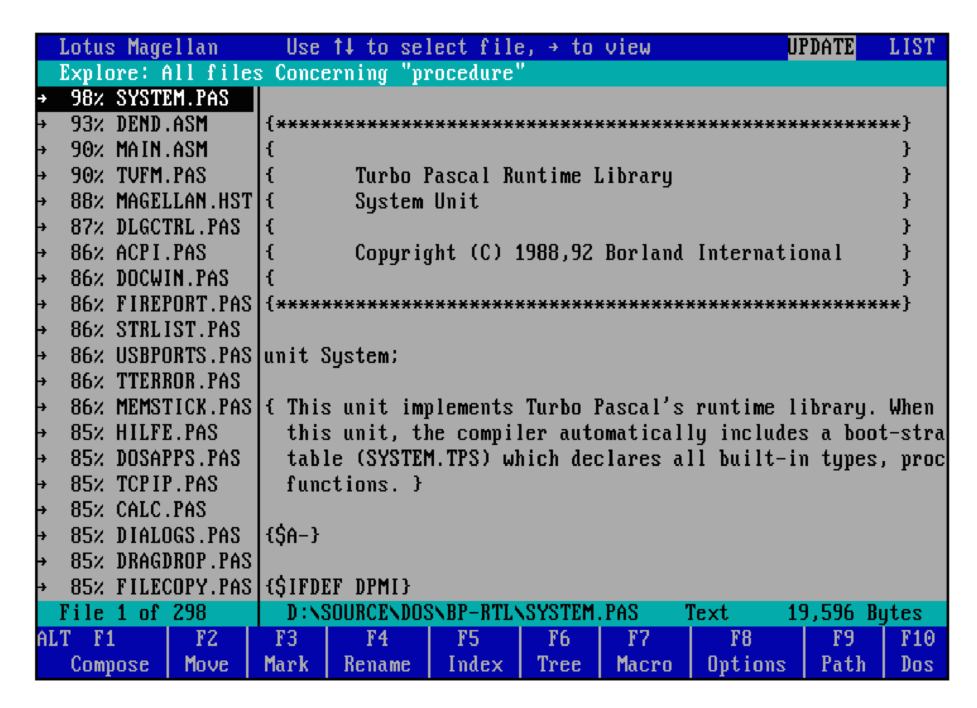


Abb. 2.1-8: Erfolgreiche Suche

Der dabei benutzte Index belegt in komprimierter Form auf der Festplatte über 21 MB, das sind etwa 2% der ursprünglichen Datenmenge. Die unterschiedlichen Dateien enthalten zudem Namen und Dateizeit aller indexierten Dateien, um Veränderungen feststellen zu können.

2.1.2 Steuerung der Indexierung

Lotus Magellan setzt einen Volltext-Index ein, der vom Benutzer konfiguriert und an unterschiedliche Sprachen angepasst werden kann. Die Konfigurationsdatei **MAGELLAN.SYM** enthält drei Tabellen (**SEPARATORS**, **CASE** und **SORT**), welche die Indexierung wesentlich beeinflussen:

```
SEPARATORS          CASE                SORT
................    ................    ................
................    ................    ................
...SSSS.........    ................    ................
0000000000......    ................    ................
.AAAAAAAAAAAAAAA    .ABCDEFGHIJKLMNO    ................
AAAAAAAAAAA....A    PQRSTUVWXYZ.....    ................
.AAAAAAAAAAAAAAA    .ABCDEFGHIJKLMNO    .ABCDEFGHIJKLMNO
AAAAAAAAAAA.....    PQRSTUVWXYZ.....    PQRSTUVWXYZ.....
AAAAAAAAAAAAAAAA    ÇÜÉÂÄÀÅÇÊËÈÏÎÌÄÅ    CUEAAAACEEEIIIAA
AAAAAAAAAAAASA.S    ÉÆÆÔÖÒÛÙYÖÜØ.Ø..    EAAOOOUUYOUO$O.$
AAAAAA..........    ÁÍÓÚÑÑ..........    AIOUNN..?....!""
.....AAAA.....S.    .....ÁÂÀ........    .....AAA.....$$.
......AA........    ......ÃÃ........    ......AA.......$
AAAAAAAAA.....A.    ÐÐÊËÈ.ÍÎÏ.....Ì.    DDEEEIIII.....I.
AAAAAA.AAAAAAA..    Ó.ÔÒÕÕ.ÞÞÚÛÙÝÝ..    OSOOOO.Þ.UUUYY..
....SS..........    ................    ................
```

Abb. 2.1-9: Wesentlicher Inhalt von ***MAGELLAN.SYM***

Das Layout der Tabellen orientiert sich am Standard-Zeichensatz mit 256 Zeichen (8 Bit), angeordnet als 16×16-Matrix. Die Tabelle **SEPARATORS** definiert für jedes Zeichen, welche Bedeutung es für die Aufteilung der Daten in einzelne Begriffe besitzen soll:

	Effekt auf die Indexierung
S	Das Zeichen trennt ein Wort und wird als alleinstehendes Symbol indexiert
A	Das Zeichen ist Teil eines Wortes
0	Das Zeichen ist Teil einer Zahl und wird als Nummer indexiert
.	Das Zeichen trennt ein Wort, wird aber nicht indexiert

*Abb. 2.1-10: Wertebereich der **SEPERATORS**-Tabelle*

Die **CASE**-Tabelle definiert für alle lateinischen Buchstaben und Umlaute bzw. internationalen Zeichen den zugehörigen Großbuchstaben, so dass alle Begriffe unabhängig von Groß- und Kleinschreibung indexiert werden. Ähnlich wird die **SORT**-Tabelle verwendet, um eine geeignete Sortierung aller Begriffe zu ermöglichen. Eine Suche nach bestimmten Attributen (etwa »Künstler«) ist nicht möglich, da der Ursprung der einzelnen Begriffe nicht erfasst wird.

2.2 SFS und MetaFS

Unix-Betriebssysteme können mehrere unterschiedliche Dateisysteme mounten; darunter auch das »Network File System« (NFS) **[RFC1094]**. Gifford hat mit dem »Semantic File System« (SFS) 1991 einen NFS-Server implementiert, der dem Client ein semantisches Dateisystem liefert **[Gif91]**.

Analog zu Lotus Magellan (→ *Kap. 2.1*) verwendet SFS Module zur Extraktion und Indexierung von Metadaten aus verschiedenen Dateiformaten; diese Module werden in **[Gif91]** »Transducer« genannt. Damit auch ältere Applikationen auf das Semantic File System zugreifen können, wird es noch unterhalb der Betriebssystem-Ebene eingebunden:

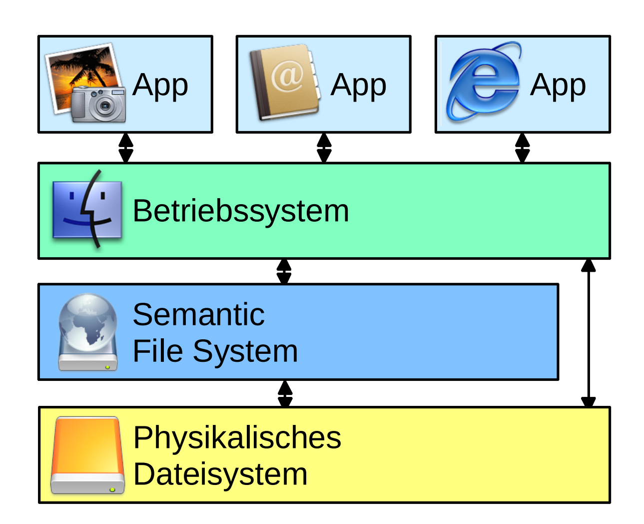


Abb. 2.2-1: Architektur des Semantic File Systems **[Gif91]**

SFS benutzt virtuelle Dateien bzw. Verzeichnisse, um Suchanfragen entgegenzunehmen und Ergebnisse zurückzuliefern. Nachfolgend wird davon ausgegangen, dass der *Mountpoint* des Semantic File System **/sfs** ist. Suchanfragen werden durch Zugriffe auf bestimmte Unterverzeichnisse realisiert. Die Metadaten werden im Gegensatz zu vielen anderen Systemen (→ *Kap. 2.1*, → *Kap. 2.10.4*) getrennt nach Attribut (Absender, Empfänger, Datum usw.) indexiert, so dass für jedes Attribut ein virtuelles Verzeichnis gebildet werden kann. Der Inhalt eines solchen Verzeichnisses sind weitere Unterverzeichnisse, die alle indexierten Attributwerte enthalten:

```
% ls -F /sfs/owner:
jones/              root/              smith/
%
```

Abb. 2.2-2: Inhalt von ***/sfs/owner*** **[Gif91]**

Ein Unterverzeichnis enthält symbolische Links zu den eigentlichen Dateien, die sich in einem physikalischen Dateisystem befinden. Diese Liste wird dabei jedes Mal dynamisch erzeugt:

```
% ls -F /sfs/owner:/smith
bio.txt@            paper.tex@          prop.tex@
%
```

Abb. 2.2-3: Alle Dateien von »Smith« **[Gif91]**

Mehrere Auswahlkriterien können in einem einzigen Pfadnamen zusammengefasst werden, das Ergebnis ist dann eine **AND**-Verknüpfung der einzelnen Filter:

```
% ls -F /sfs/owner:/smith/text:/resume
bio.tex@
%
```

Abb. 2.2-4: Alle Dateien von »Smith«, die den Begriff »resume« enthalten **[Gif91]**

Das virtuelle Verzeichnis **field:** zeigt alle verfügbaren Attributnamen als Unterverzeichnisse an. Das Verzeichnis **text:** gestattet dabei das Durchsuchen aller Dateikörper, was das Schema der einzelnen Dateiformate durchbricht:

```
% ls -F /sfs/field:
author:/          exports:/         owner:/           text:/
category:/        ext:/             priority:/        title:/
date:/            imports:/         subject:/         type:/
dir:/             name/:
%
```

Abb. 2.2-5: Virtuelles Verzeichnis ***field:*** **[Gif91]**

2.2.1 Weiterentwicklungen

Die Idee, Verzeichnisse dynamisch und auf den Attributen bzw. dem Inhalt von Dateien basierend zu erzeugen, wurde u.a. 1999 von Gopal et al. wieder aufgegriffen **[Gop99]**, die sich explizit auf das SFS aus **[Gif91]** berufen. Während SFS alle dynamisch erzeugten Verzeichnisse unter dem Mountpoint **/sfs** erzeugt, vermischt das in **[Gop99]** entworfene Dateisystem HAC (für »Hierarchy And Content«) die Verzeichnisstruktur des physikalischen Dateisystems mit virtuellen Ordnern.

Das 2005 von Mills in **[Mil05]** veröffentlichte MetaFS ist weitgehend identisch mit SFS, unterstützt allerdings moderne Dateiformate wie MP3 und JPEG sowie deren Standards für Metadaten **[Nil05]** **[EXI07]**. Darüber hinaus enthält MetaFS zusätzliche Befehle zur Manipulation der verschiedenen Attribute.

2.2.2 Performanz

In **[Gif91]** wird auch die Leistungsfähigkeit des Semantic File System untersucht. Da der Prototyp keine Unterstützung für das Umbenennen oder Löschen von Dateien bietet, wird eine Liste mit gelöschten Dateien verwaltet, die bei Suchanfragen aus dem Suchergebnis entfernt werden. Die Einträge im eigentlichen Index werden erst bei einer vollständigen Neuindexierung gelöscht.

[Gif91] beschreibt eine Neuindexierung von Testdaten, die am 23. Juli 1991 vorgenommen wurde. An diesem Tag war die Gesamtgröße des Dateisystems 326 MB, wobei 230 MB von öffentlich lesbaren Dateien belegt wurden. Nur 68 MB konnten indexiert werden, da für die übrigen 162 MB keine Übersetzer (»transducer« genannt) verfügbar waren. Die Dateien sind wie folgt klassifiziert:

	Anzahl	Größe
Object	871	8.503 KB
Quelltext	2.755	17.991 KB
Text	1.871	20.638 KB
Andere	2.274	21.187 KB
	7.771	68.319 KB

Abb. 2.2-6: Indexierte Dateien am 23. Juli 1991 **[Gif91]**

Der erstellte Index hatte eine Größe von 10019 KB, wobei 6621 KB von Tabellen und 3398 KB durch die Baumstruktur belegt wurden. Der Index belegt also ein Siebtel der ursprünglichen Datenmenge. Die Zeit für die vollständige Neuindexierung von einer Million Attributwerten, die in den Dateien enthalten waren, betrug 96 Minuten **[Gif91]**:

	Zeit
Verzeichnisbaum lesen	0:07
Dateityp ermitteln	0:01
Transducer »Verzeichnis«	0:01
Transducer »Object«	0:08
Transducer »Quelltext«	0:23
Transducer »Text«	0:23
Transducer für andere Dateitypen	0:24
Indextabellen erzeugen	1:22
Indexbaum erzeugen	0:06
	1:36

Abb. 2.2-7: Zeitbedarf für vollständige Neuindexierung am 23. Juli 1991 **[Gif91]**

[Gif91] bemerkt, dass das Erstellen des Index maßgeblich von der I/O-Geschwindigkeit abhängt, da die CPU während 60% der Zeit unbelastet war. Auch die Bearbeitung von Suchanfragen hängt stark von der Datenträgergeschwindigkeit ab. Die Zeit, die von Lotus Magellan (→ *Kap. 2.1*) für eine vollständige Neuindexierung benötigt wird, ist mit dem Zeitbedarf in **[Gif91]** vergleichbar.

2.3 Rufus

Das Rufus-Projekt besteht aus den Applikationen **xrufus** (grafische Applikation), **rufustrn** (erweiterter Usenet-Reader) und **rufusbld** (Indexierung), die alle auf Userebene ablaufen **[Sho93]**. Damit entspricht das Rufus-Design prinzipiell dem einer Desktop-Suchmaschine (→ *Abb. 2.1-1*).

Ähnlich wie das Semantic File System (→ *Kap. 2.2*) zielt das Rufus-Projekt darauf ab, das Auffinden von Dateien zu verbessern. Rufus bietet dazu nicht nur eine einfache Suchfunktion, sondern bettet aufgefundene Dateien in ein objektorientiertes Konzept ein **[Sho93]**.

In **[Sho93]** wird festgestellt, dass ein ideales Dateisystem die Semantik der Dateiformate kennt, also ein semantisches Dateisystem sein sollte. Für jeden Dateityp kann dann nicht nur eine Suchfunktion bereitgestellt, sondern auch automatisch die geeignete Applikationen zur Bearbeitung aufgerufen werden. Um dies zu erreichen, wird in **[Sho93]** ein Klassifizierer eingeführt, der die Klasse (also Format bzw. Typ) einer gegebenen Datei ermittelt. Die erzeugten Objekte extrahieren die Attribute der Dateien und sind somit ein wichtiger Bestandteil der Suchfunktion, die ihrerseits einen Index über alle erkannten Attribute nutzt.

2.3.1 Klassifizierer

Eine zuverlässige Klassifizierung einer Datei ist notwendig, damit das richtige Modul die Metadaten zur Indexierung extrahieren kann. Jede Rufus-Klasse besitzt eine Funktion, die einen Wert zwi-

schen 0 und 10 zurückliefert, abhängig von der wahrscheinlichen Zugehörigkeit der untersuchten Datei zur Klasse. Dabei sucht ein Klassifizierer nach bestimmten Schlüsselworten oder Binärmustern, die typisch für ein bestimmtes Dateiformat sind. Als Test wurden 847 Beispieldateien verschiedener Typen klassifiziert, davon jedoch nur 90% korrekt **[Sho93]**.

2.3.2 Objekte

Basierend auf der jeweiligen Klasse wird für jede indexierte Datei ein Objekt erstellt, das neben den extrahierten Metadaten auch spezifische Methoden enthält, beispielsweise zur Anzeige, zur Bearbeitung oder zum Drucken. Rufus-Objekte sind allerdings nicht in der Lage, Änderungen (etwa ergänzte Metadaten) auch an den Dateien im physikalischen Dateisystem durchzuführen **[Sho93]**.

2.4 Microsoft OLE

OLE ist die Abkürzung für »Object Linking and Embedding« und bezeichnet die Objektorientierung von Applikationen und ihren Dateiformaten, die 1993 mit Windows 3.1 veröffentlicht wurde. OLE ist somit etwa gleichzeitig mit Rufus (→ *Kap. 2.3*) entwickelt worden, und bettet wie Rufus Dateien in ein objektorientiertes Konzept ein. Microsoft OLE zielt allerdings auf die Zusammenarbeit verschiedener Programme ab und löst damit im Rahmen dieser Arbeit nur ein Teilproblem.

Ein Objekt wird als Menge von Informationen definiert, die entweder verknüpft (»linked«) oder in ein anderes Dokument eingebettet (»embedded«) werden. Ein eingebettetes Objekt ist eine ins Hauptdokument aufgenommene Kopie einer Datei, die mit einer anderen Anwendung erstellt wurde. Ein eingebettetes Objekt ist nicht mehr mit dem Original verbunden; wird das Objekt also geändert, wirken sich diese Änderungen nicht auf die Quelldatei aus und umgekehrt **[Mic93]**.

Wird im Gegensatz dazu ein verknüpftes Objekt erstellt, so wird eine Verbindung zwischen dem Zieldokument und dem Quelldokument erschaffen. Obwohl das verknüpfte Dokument als Teil der Zieldatei angezeigt und auch ausgedruckt wird, sind die Daten, aus denen das Objekt besteht, weiterhin nur im Quelldokument vorhanden. Wird ein verknüpftes Objekt modifiziert, ändert sich gleichzeitig das Quelldokument. Umgekehrt wirken sich auch Änderungen des Quelldokuments auf das verknüpfte Objekt im Zieldokument aus (es wird aktualisiert) **[Mic93]**.

Das Verknüpfen und Einbetten von Objekten ist dem Kopieren und Einfügen sehr ähnlich. Das Ergebnis hängt davon ab, mit welcher Software gearbeitet wird. Wenn die Anwendung OLE unterstützt, wird ein verknüpftes oder eingebettetes Objekt erstellt. Andernfalls wird nur eine statische Kopie der Quelldatei erzeugt **[Mic93]**.

Microsoft OLE greift nicht tief ins Betriebssystem ein, sondern stellt für OLE-kompatible Anwendungen lediglich eine Registrierung aller anderen OLE-Anwendungen (→ *Kap. 2.4.2*) und ein entsprechendes API zur Verfügung. Mit dem Objekt-Manager (→ *Kap. 2.4.3*) steht außerdem ein Hilfsprogramm zur Verfügung:

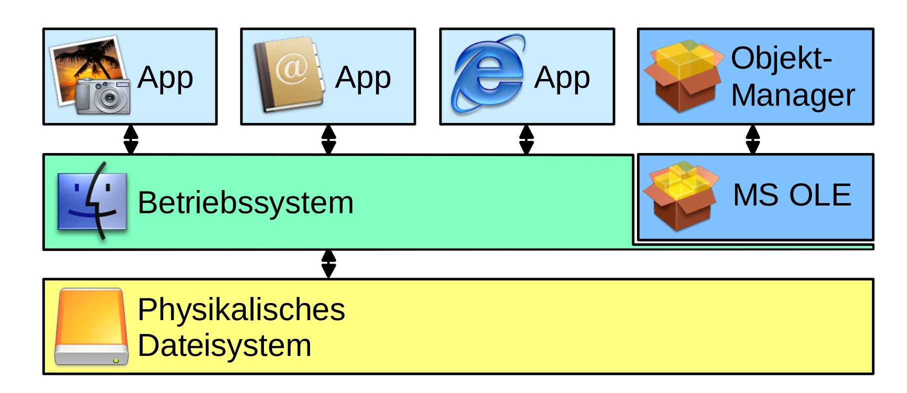


Abb. 2.4-1: OLE-Architektur

2.4.1 Beispiel

Als Zieldatei soll ein einfaches Word-Dokument dienen. Microsoft Word verfügt natürlich über OLE-Fähigkeiten:

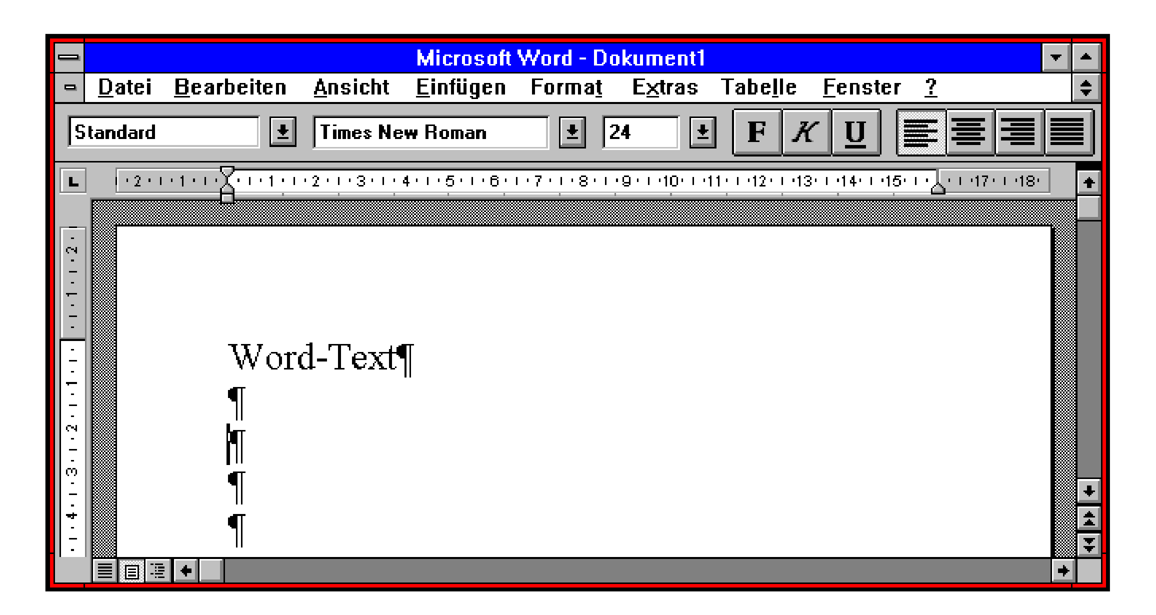


Abb. 2.4-2: Einfaches Word-Dokument

In das Dokument kann nun ein OLE-Objekt eingefügt werden. Entweder wird dazu ein neues Quelldokument erstellt und als Objekt eingebunden, oder das Objekt wird aus einer bereits gespeicherten Datei erzeugt:

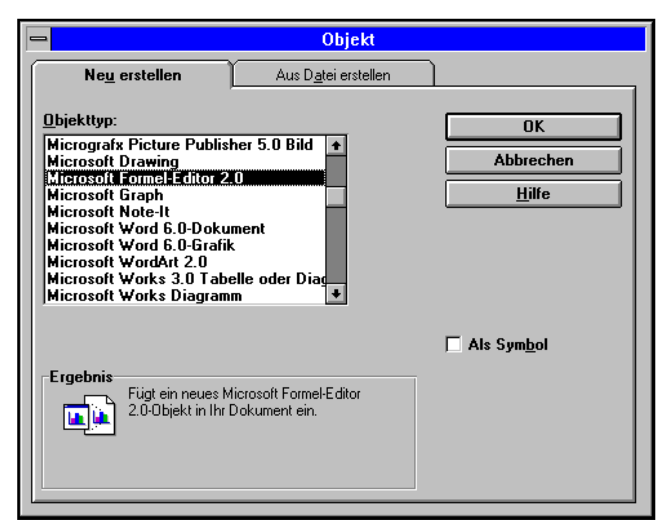


Abb. 2.4-3: *Neues Objekt erstellen*



Abb. 2.4-4: *Objekt aus einer Datei erzeugen*

OLE wird nicht nur von großen Applikationen unterstützt, sondern ist vor allem für Tools nützlich. Zusammen mit MS Word werden einige kleinere Programme mitgeliefert, die OLE-Objekte erstellen können: u.a. MS Draw (Vektorgrafiken), MS Formel-Editor, MS Graph (Diagramme) und MS WordArt (Texteffekte). In diesem Beispiel wird ein Formel-Objekt eingefügt; der Formel-Editor benutzt ab OLE 2.0 das Word-Fenster, um sich selbst innerhalb des Zieldokuments darzustellen:

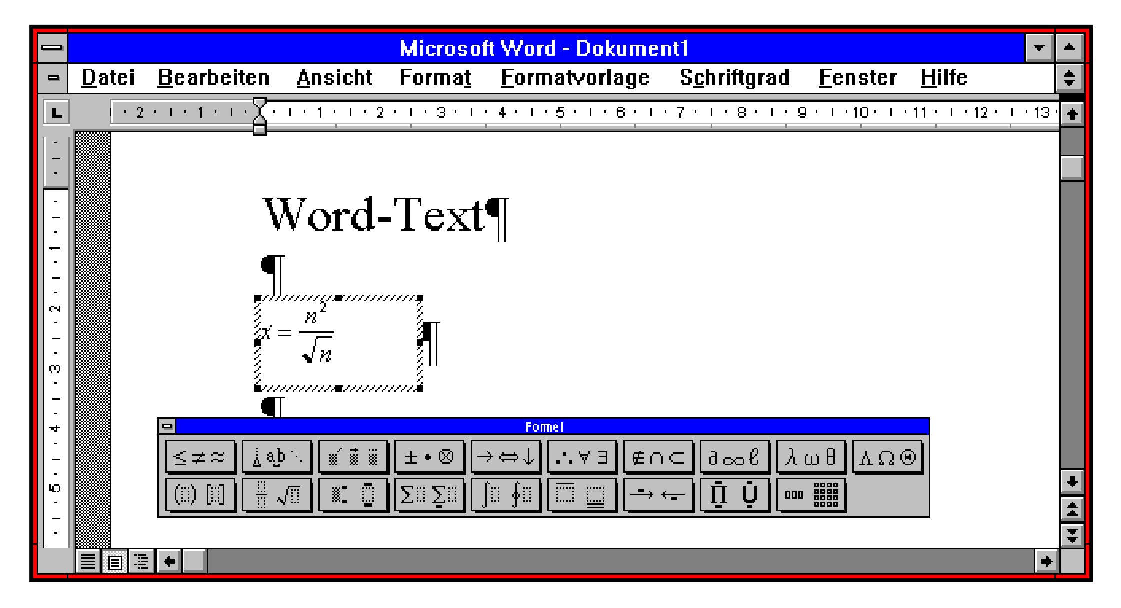


Abb. 2.4-5: *Formel-Editor innerhalb des Word-Fensters*

Wenn der Formel-Editor beendet wird, übernimmt die Ziel-Applikation, also Microsoft Word, wieder die Kontrolle über das Fenster. Das eingebettete Objekt wird nun innerhalb des Dokuments dargestellt und kann auch ausgedruckt werden:

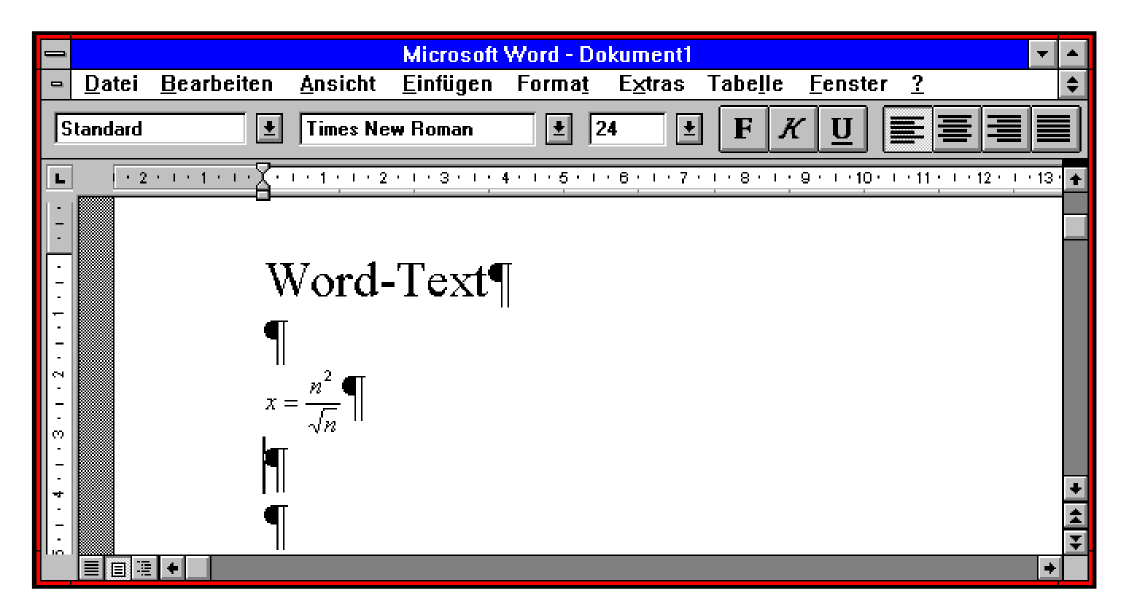


Abb. 2.4-6: Eingebettetes Formel-Objekt

Wäre der Formel-Editor getrennt gestartet und die Formel als Datei gespeichert worden, hätte die Datei als verknüpftes Objekt ins Zieldokument eingefügt werden können (→ *Abb. 2.4-4*).

2.4.2 Registry

Damit das Erstellen und Einfügen von Objekten möglich ist, muss jedes Objekt einem Anwendungsprogramm zugeordnet werden. Microsoft OLE benutzt dazu keinen Klassifizierer (→ *Kap. 2.3.1*), sondern verwaltet in der Datei **REG.DAT** im **WINDOWS**-Verzeichnis eine Registrierung, in die sich OLE-kompatible Programme während der Installation eintragen:

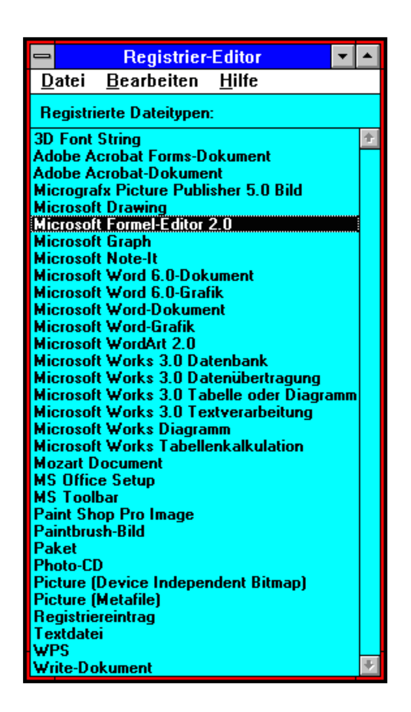


Abb. 2.4-7: Registry unter Windows 3.11

Zu jedem Datentyp werden wichtige Informationen gespeichert, vor allem mögliche Befehle (**print** und **open** in der Gruppe **shell**) sowie die zuständige Programmdatei (**EQNEDIT.EXE**):

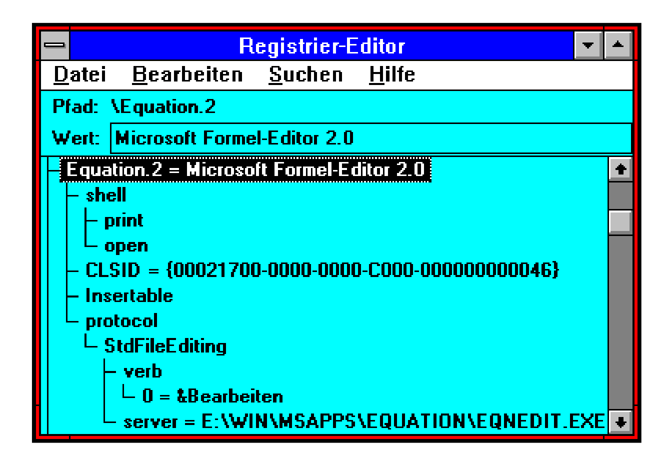


Abb. 2.4-8: Registrierung des Formel-Editors

Ab Windows 95 wurde die Registry aufgewertet und dient als zentrale Konfigurationsdatei für beinahe alle Programme und Einstellungen; sie kann immer noch mit **REGEDIT.EXE** bearbeitet werden, und sogar die Angaben über Dateiformate und OLE-kompatible Programme sind noch vorhanden:

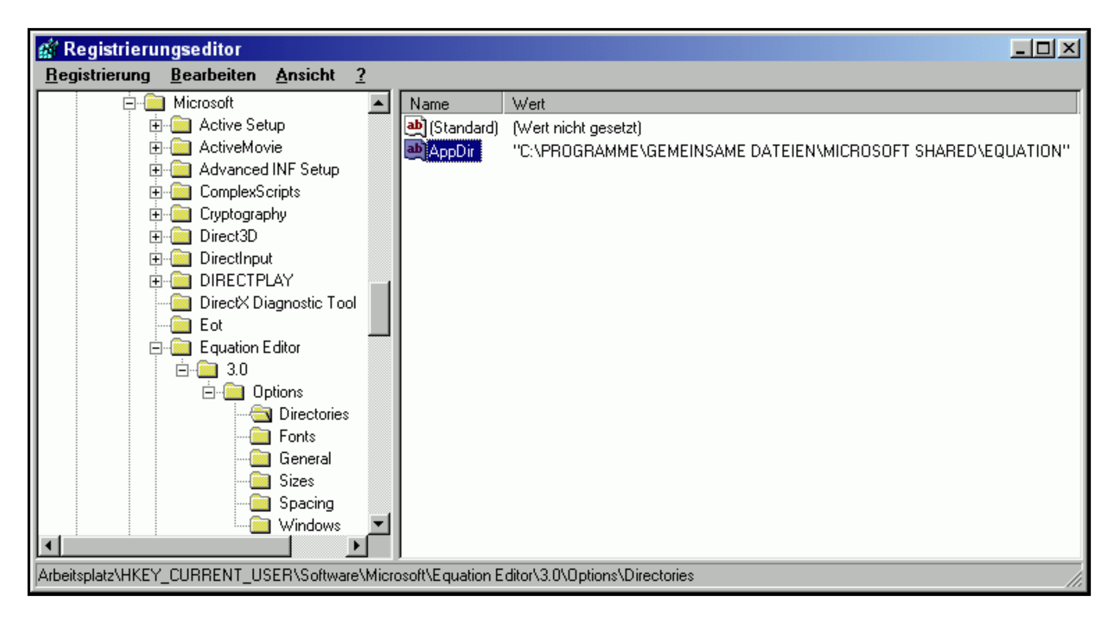


Abb. 2.4-9: Registrierungs-Editor ab Windows 95

2.4.3 Objekt-Manager

Der Objekt-Manager ist ein Systemprogramm und dient dazu, ein Dokument einer nicht OLE-fähigen Anwendung als Symbol einzufügen. Dazu wird aus den Ursprungsdaten ein OLE-Paket erzeugt. Es handelt sich dabei um ein eigenständiges Objektformat, das mit der Programmdatei des Objekt-Managers registriert wurde (**PACKAGER.EXE**):

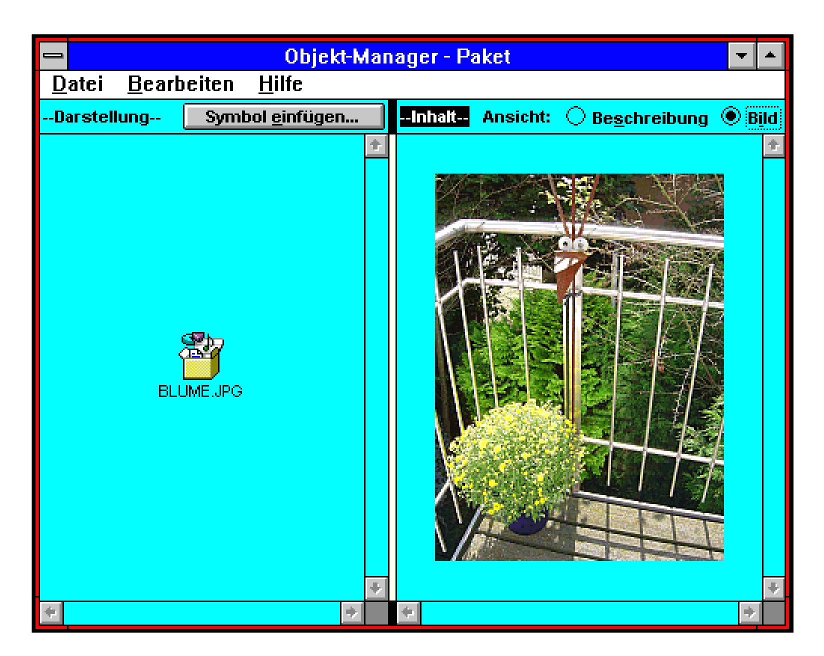


Abb. 2.4-10: Paket-Erstellung mit dem Objekt-Manager

Das erstellte Paket kann danach in ein Zieldokument eingebettet werden; Verknüpfungen sind nicht möglich. Im obigen Beispiel (→ *Kap. 2.4.1*) würde statt des eigentlichen Bildes also nur ein Symbol innerhalb des Word-Dokuments erscheinen. Das Paket wird also ähnlich dargestellt wie eine an eine EMail angehängte Datei.

2.5 Microsoft Indexerstellung

Zusammen mit Microsoft Office 95 und 97 wird das Program »Indexerstellung« installiert und fortan im Hintergrund ausgeführt. Das Programm indexiert im Hintergrund alle Dateien, die zu MS Office kompatibel sind. In der Systemsteuerung erscheint ein neues Kontrollfeld, mit dem sich die Indexerstellung konfigurieren lässt:



Abb. 2.5-1: Indexerstellung in der Systemsteuerung

Der erstellte Index wird von den Office-Programmen (Word, Excel, PowerPoint usw.) benutzt, um beim Öffnen von Dateien eine inhaltsbezogene Suche anzubieten:

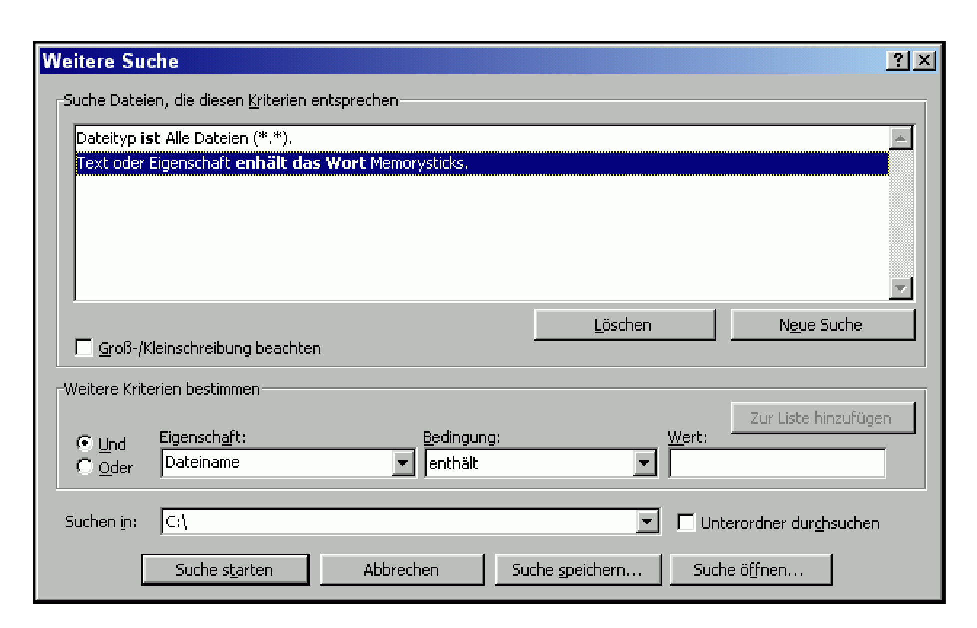


Abb. 2.5-2: Inhaltsbezogene Suche in Microsoft Word 97

Damit entspricht auch die Architektur der Microsoft Indexerstellung derjenigen einer Desktop-Suchmaschine (→ *Abb. 2.1-1*). Dafür ist es unerheblich, dass die Suche selbst nur innerhalb der Office-Software stattfindet. Vor einer Suche müssen jedoch, wie bei anderen Produkten auch, alle relevanten Dateien indexiert werden. In einem einfachen Test wurden alle Office- und Web-Dokumente eines Windows-PCs, insgesamt 2067 Dateien, innerhalb von etwa 15 Minuten unter hoher Auslastung der Systemressourcen erfasst; der fertige Index belegte dabei ca. 5,1 MB.

Die starke Beanspruchung von Ressourcen ist wegen der alle 2 Stunden stattfindenden Neuindexierung bedenklich. Deshalb zog die Microsoft Indexerstellung starke Kritik von Anwendern auf sich, vielleicht auch weil von einem Office-Paket kein derartiges Verhalten erwartet wird. Durch diese Kritik hat sich Microsoft schließlich veranlasst gefühlt, im WWW unter **[Mic00]** eine Anleitung zum Deaktivieren der Indexerstellung bereitzustellen.

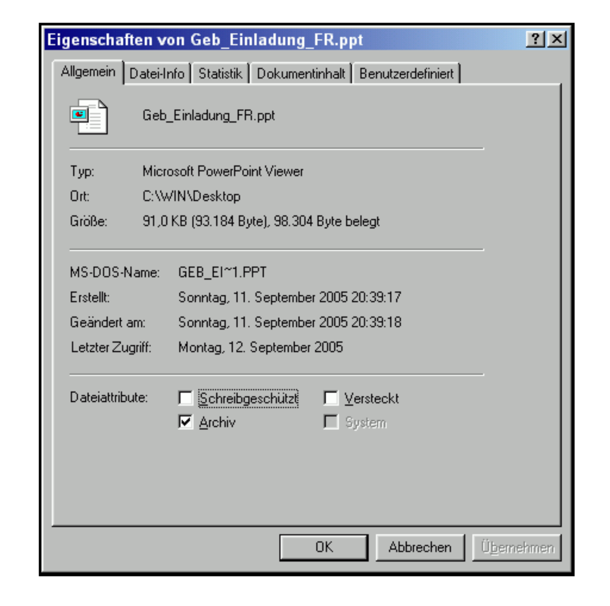
2.6 Microsoft Windows Explorer

Der »Explorer« ist seit Windows 95 der Datei-Manager des Betriebssystems und übernimmt auch die Darstellung des Desktops und der Task-Leiste, die sich typischerweise am unteren Bildschirmrand befindet. Zur Darstellung von Dateieigenschaften und zum Ändern der Attribute bietet der Explorer eine Dialogbox an, die ohne weitere Maßnahmen für alle Dateiformate gleich ist:



Abb. 2.6-1: Eigenschaften einer normalen Datei

Applikationen haben jedoch die Möglichkeit, den Explorer durch ein Plug-In zu erweitern. Vor allem die diversen Programme aus dem Office-Paket (Word, Excel, PowerPoint u.a.) machen von dieser Möglichkeit Gebrauch, aber auch der Windows Media Player. Dadurch wird der »Eigenschaften«-Dialog semantisch (also typabhängig) und zeigt abhängig vom jeweiligen Dateiformat zusätzliche Metadaten an:



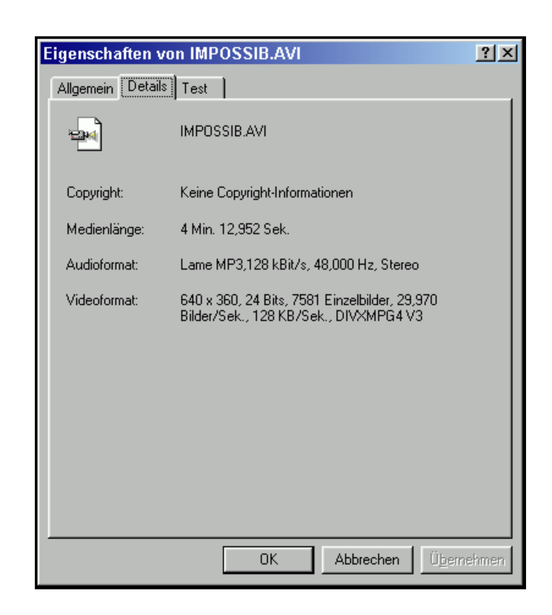




Abb. 2.6-2: Powerpoint-Datei

Abb. 2.6-3: AVI-Video

Diese Plug-Ins sind als eine Erweiterung von Microsoft OLE (→ *Kap. 2.4*) anzusehen, da sie für bestimmte Dateitypen (wie z.B. PowerPoint-Präsentationen oder AVI-Videos) in die Registry-Datei (→ *Kap. 2.4.2*) eingetragen wurden.

2.7 BeFS

Die Firma Be Inc. wurde 1991 mit dem Ziel gegründet, ein neuartiges Betriebssystem und darauf abgestimmte Hardware zu entwickeln. Eine erste Version von BeOS wurde im Sommer 1996 auf der MacWorld Expo in Boston vorgestellt. Da BeOS von Grund auf neu entwickelt wurde, schleppt das Betriebssystem keine Altlasten früherer Systeme mit sich herum und bringt außerdem viele Vorteile anderer Betriebssysteme wie Mac OS, Windows und Unix zusammen **[Har05]**.

BeOS nutzt das »BeOS File System« (BeFS) als Dateisystem. Es weist Ähnlichkeiten zum *ext2*-Dateisystem **[Kol07b]** auf, kombiniert dieses aber mit den Fähigkeiten eines semantischen Dateisystems. BeFS verfolgt somit einen integrierten Ansatz:

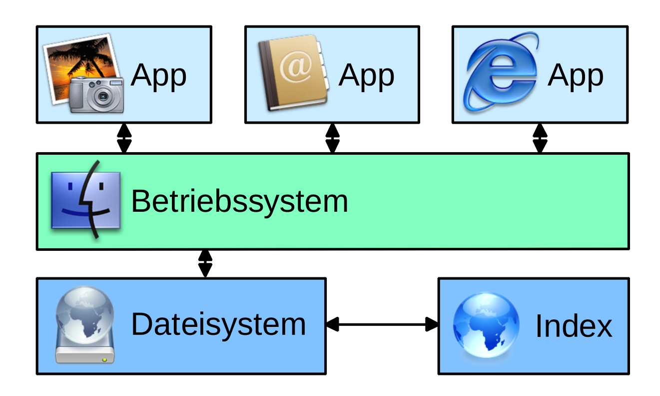


Abb. 2.7-1: Integrierte Architektur von BeOS

In diesem Abschnitt wird auf die besonderen Fähgkeiten von BeFS eingegangen: auf Übersetzer (→ *Kap. 2.7.1*), Postfächer (→ *Kap. 2.7.2*) und auf die Suchfunktion (→ *Kap. 2.7.3*).

2.7.1 Übersetzer

Unter BeOS besitzen die meisten Applikationen, wie bei anderen Betriebssystemen auch, einen Menupunkt »Speichern unter«, durch den man neben einem anderen Namen auch ein anderes Dateiformat auswählen kann. Traditionell ist es Aufgabe der Anwendung, den Programmcode zum Speichern der Datei bereitzustellen. Dadurch haben die Entwickler mehrfache Arbeit, da diese Module in vielen Programmen implementiert werden müssen; besonders problematisch ist das Fehlen eines bestimmten Formats in einer Applikation **[Hac99]**.

BeOS stellt allen Programmen Übersetzungsmodule an einer zentralen Stelle zur Verfügung, die das Laden und Speichern von Dateien übernehmen. Dadurch sind die Applikationen kleiner, und ihre Fähigkeiten können mit neuen Modulen gemeinsam erweitert werden **[Hac99]**:

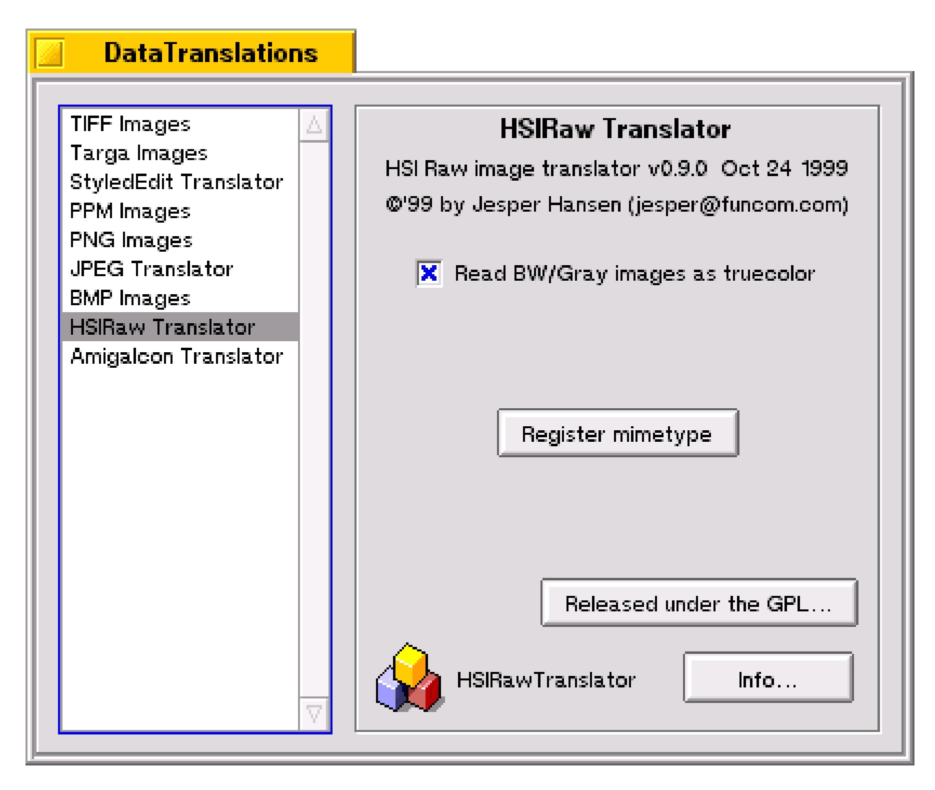


Abb. 2.7-2: BeOS-Übersetzer

Unter »Preferences« kann die Liste der installierten Übersetzer eingesehen werden. Viele Module können hier auch konfiguriert werden: beispielsweise kann beim JPEG-Übersetzer die Bildqualität für das Speichern eingestellt werden, da Bilder im JPEG-Format verlustbehaftet komprimiert werden **[Hac99]**.

2.7.2 Postfächer

Eine Besonderheit stellt auch der Umgang mit elektronischer Post dar. Während herkömmliche Betriebssysteme das Speichern der EMails einer Zusatzsoftware überlassen (→ *Kap. 4*), ist unter BeOS die entsprechende Funktionalität ins Dateisystem integriert. Die verschiedenen Ordner für den Posteingang, Postausgang und vom Benutzer angelegte Postverzeichnisse haben spezielle Attribute, die die Anzahl der ungelesenen, zu sendenden oder bereits abgeschickten EMails anzeigen. Die Attribute wie Absender, Empfänger, Thema oder Datum werden direkt als Eigenschaft der Datei angezeigt:

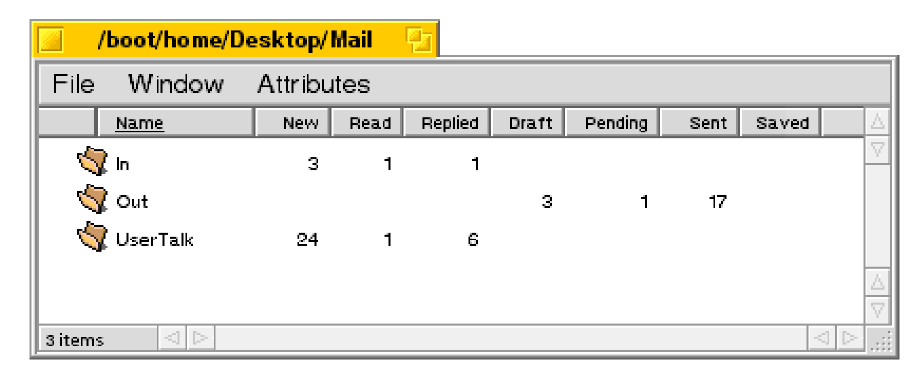


Abb. 2.7-3: Postfächer

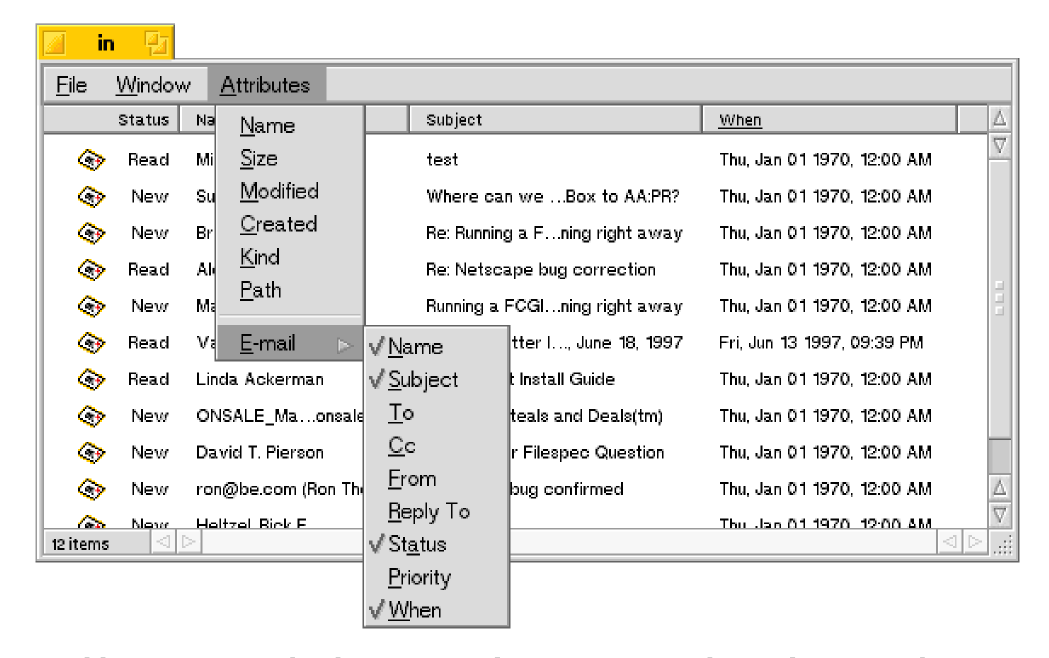


Abb. 2.7-4: EMails als eigenständige Dateien mit besonderen Attributen

Das BeOS-Dateisystem bindet die besonderen Attribute von Maildateien ein, so dass es Aspekte eines semantischen Dateisystems besitzt. Da das Dateisystem aber auch nach wie vor für die physikalische Speicherung in einzelnen Blöcken sorgt, stellt BeFS eine Mischform dar. Detaillierte Informationen zum BeOS-Dateisystem, insbesondere auch zum physikalischen Teil und zur Performanz, finden sich in **[Gia99]**.

2.7.3 Suchfunktion

Da BeFS die besonderen Attribute einiger Dateiformate einbindet, ist es einfach, eine inhaltsbezogene Suche über diese Attribute zu implementieren. Natürlich ist eine einfache Suche nach Dateien mit einem bestimmten Namen oder Namensteil möglich (»All files and folders« und »by Name«, → *Abb. 2.7-5*). Zusätzlich kann die Suche auf bestimmte Dateitypen eingegrenzt werden, da BeOS das Dateiformat unabhängig von einer möglichen Namensendung verwaltet (»E-mail«, → *Abb. 2.7-6*). Eine Suche nach Dateien mit bestimmten Attributen (»by Attribute«, → *Abb. 2.7-6*)

ist aber ebenso möglich. Wird also beispielsweise nach Maildateien gesucht, können alle bei diesem Dateityp vorhandenen Metadaten in die Suchanfrage aufgenommen werden (→ *Abb. 2.7-7*).

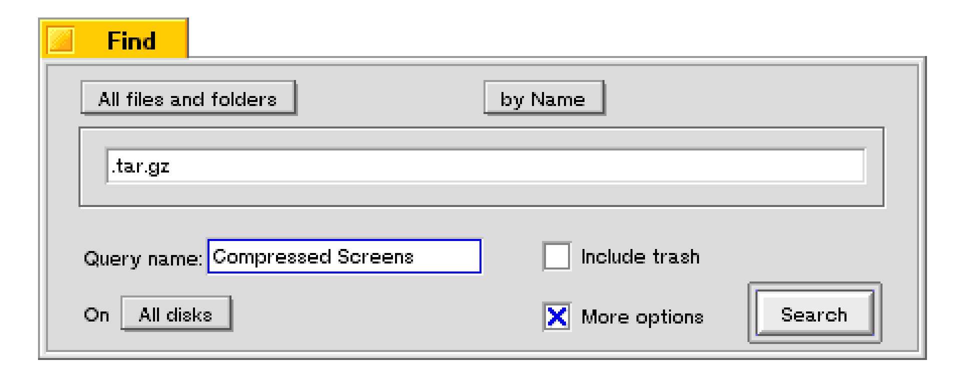


Abb. 2.7-5: Suche nach Dateien anhand des Namens bzw. der Endung

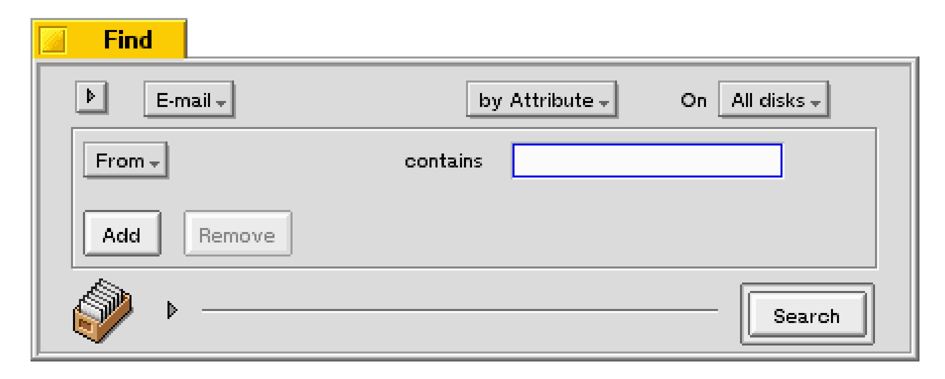


Abb. 2.7-6: Suche nach Mails

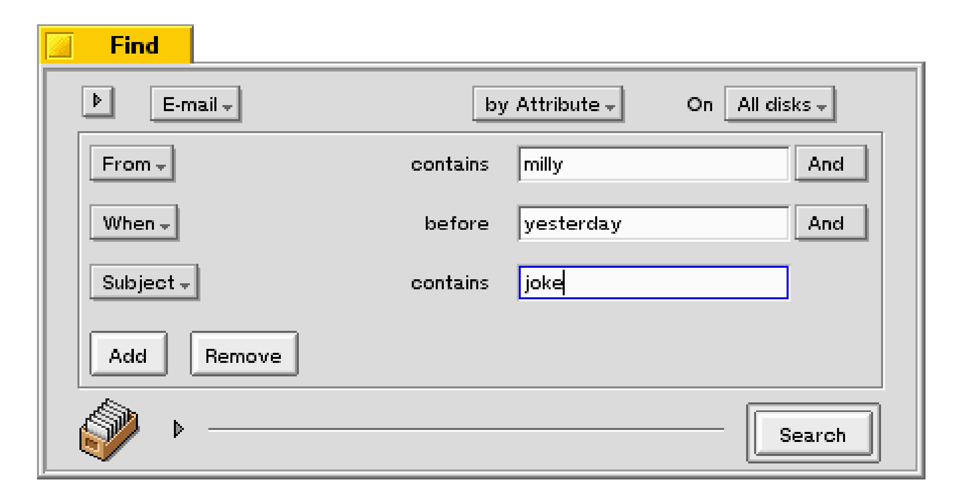


Abb. 2.7-7: Suche nach EMails anhand typspezifischer Attribute

2.8 DBFS

Das »Database File System« (DBFS) wurde als Linux-Addon im August 2004 von Gorter im Rahmen seiner Diplomarbeit an der Universität Twente implementiert **[Gor04]**. Bei DBFS handelt es sich um ein semantisches Dateisystem, dessen Ziel die Überwindung der klassischen hierarchischen

Verzeichnisstruktur ist. Die Software greift tief ins Linux-System und die KDE-Oberfläche **[KDE06]** ein, um sich nahtlos in bereits bestehende Applikationen zu integrieren. Die Architektur wirkt dementsprechend komplex:

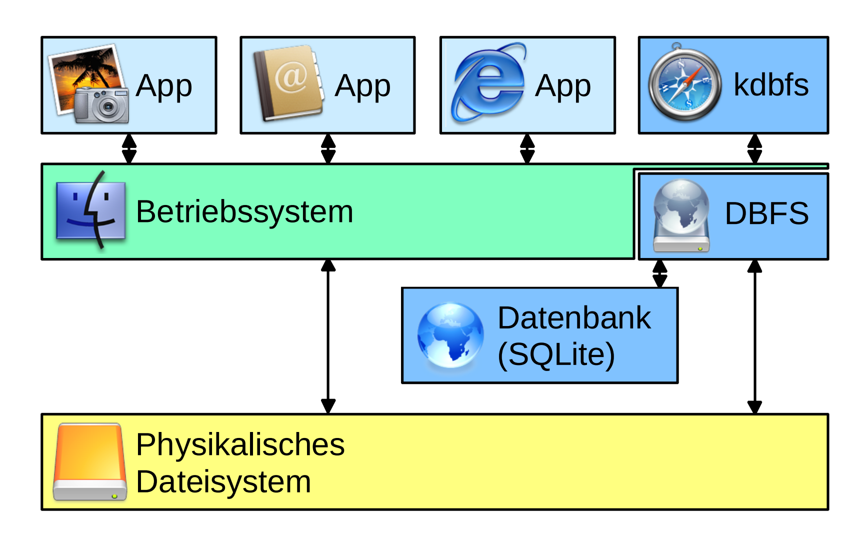


Abb. 2.8-1: Architektur des DBFS

Das DBFS schaltet sich als semantisches Dateisystem zwischen das physikalische Dateisystem und das Betriebssystem ein. Einzelne Module der KDE-Oberfläche werden dabei ersetzt, so dass alle KDE-Applikationen ein verändertes Dialogfenster für das Öffnen und Speichern von Dateien aufrufen:

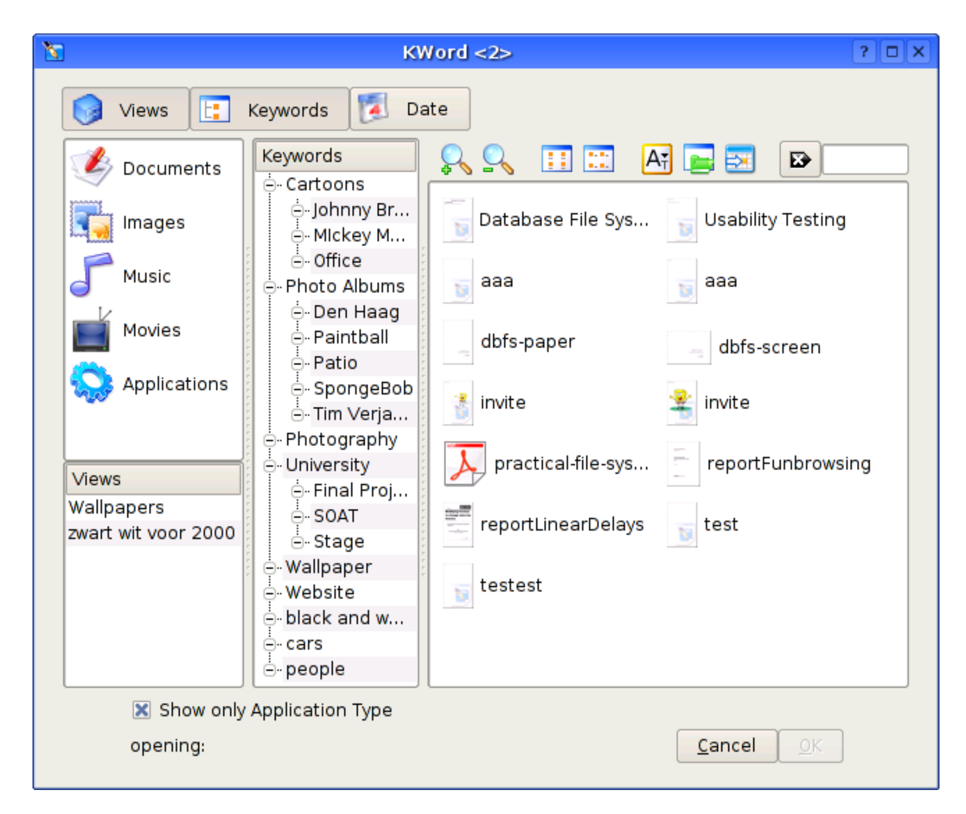


*Abb. 2.8-2: Öffnen einer Datei mit **kword** bei installiertem DBFS* **[Gor04]**

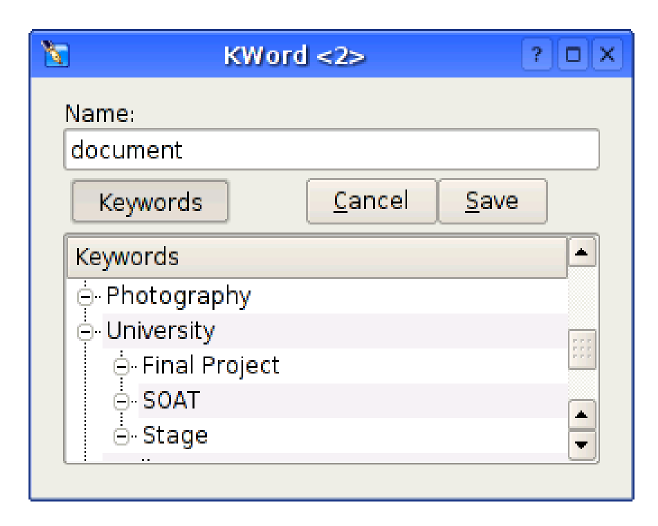


*Abb. 2.8-3: Speichern einer Datei mit **kword** bei installiertem DBFS* **[Gor04]**

Als Ergänzung zum KDE-Dateimanager dient das Programm **kdbfs**, das in wesentlichen Teilen dem »Öffnen«-Dialog entspricht:

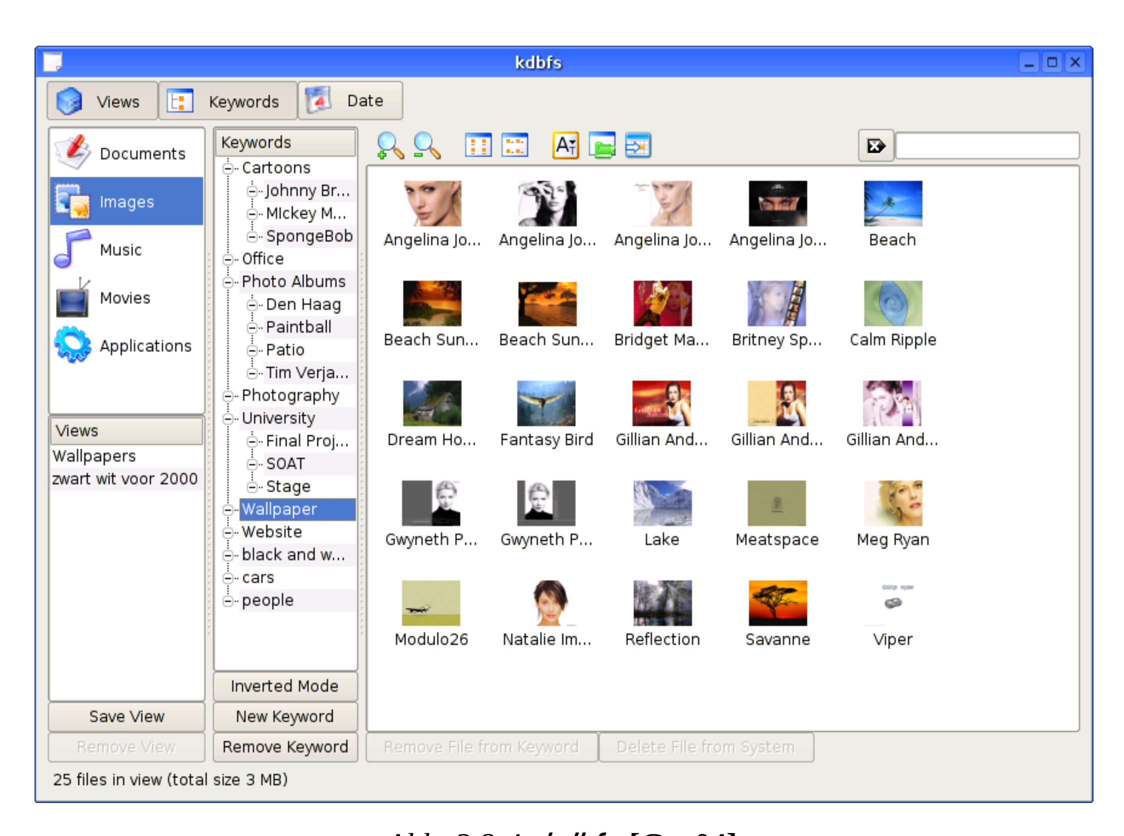


*Abb. 2.8-4: **kdbfs*** **[Gor04]**

Intern benötigt DBFS die Unterstützung einer *SQL*-kompatiblen Datenbank, um einen Index zu verwalten **[Gor04]**. DBFS speichert in diesem Index allerdings keine Metadaten aus Dateien ab, sondern setzt beim Speichern auf die Eingabe von Schlüsselworten durch den Benutzer (→ *Abb. 2.8-3*). Das DBFS erfordert also genauso wie Verzeichnisbäume die aktive Mithilfe des Benutzers, der einen entsprechenden Aufwand betreiben muss **[Mil05]**.

2.9 Microsoft WinFS

»WinFS« ist die Abkürzung für »Windows Future Storage« und sollte ursprünglich mit Windows Vista veröffentlicht werden, bis das Projekt auf unbestimmte Zeit verschoben und schließlich im Juni 2006 eingestellt wurde **[Wik05]**. Am 29.08.2005 wurde jedoch eine erste Beta-Version von Microsoft in Umlauf gebracht, die unter Windows XP lauffähig ist **[Mic05]**:



Abb. 2.9-1: WinFS Beta 1 **[Mic05]**

WinFS verwaltet keine Dateien aus dem physikalischen Dateisystem, sondern speichert Dateien über den Microsoft SQL Server 2005 (→ *Kap. 3.3.5.1*) in einer *BLOB*-relationalen Datenbank (→ *Kap. 4.2.1*) ab. Um die Kompatibilität zu älteren Anwendungen zu gewährleisten, bildet ein Modul von WinFS diese Datenbank ins Dateisystem ab:

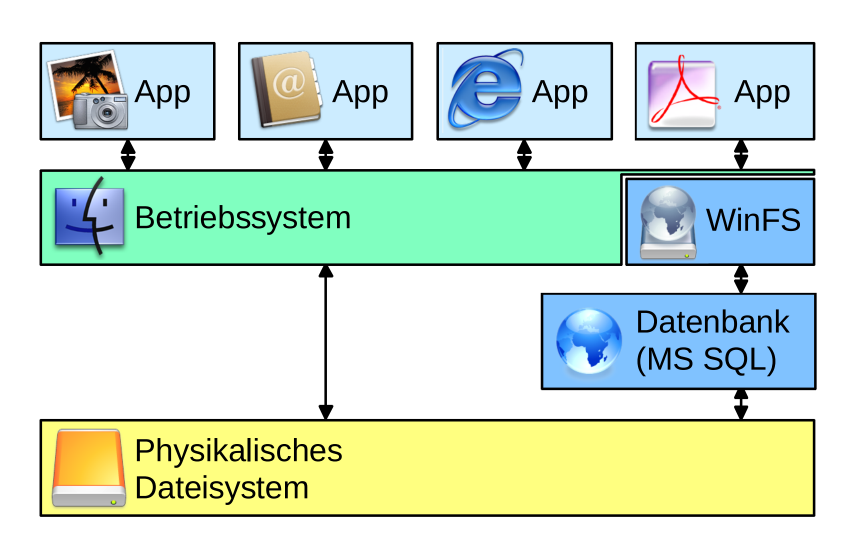


Abb. 2.9-2: Integration von WinFS in Microsoft Windows

Nach der Installation von WinFS erscheint im »Arbeitsplatz«-Fenster das neue Symbol »WinFS Stores« **[Mic08c]**:

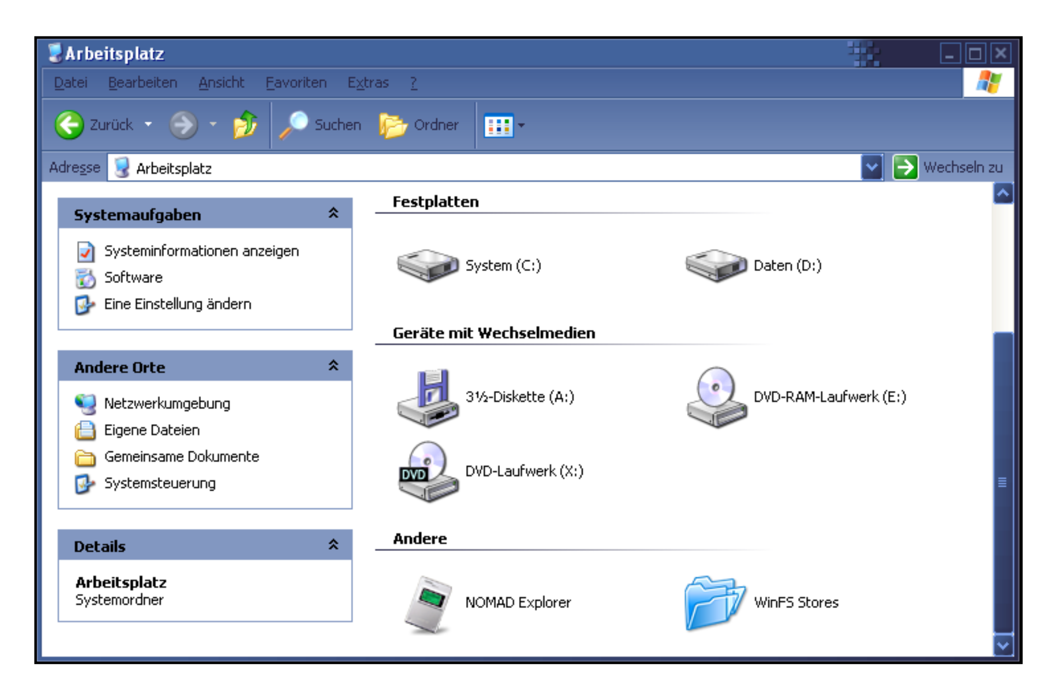


Abb. 2.9-3: Einbindung von WinFS in den Explorer

WinFS kann mehrere unabhängige Datenbanken (»Stores«) verwalten, z.B. eine für jeden Benutzer. Beim Zugriff über den Windows-Explorer verhält sich ein Store wie eine normales Laufwerk mit Dateien und Verzeichnissen, da WinFS hier abwärtskompatibel ist. Die eigentlichen Neuerungen von WinFS werden erst beim nativen Zugriff sichtbar. Das Hilfsprogramm »WinFS Type Browser« dient der Spezifikation von Dateiformaten, der »StoreSpy« ist ein rudimentärer Datei-Manager, der im Gegensatz zum Explorer direkt auf WinFS zugreift.

2.9.1 WinFS Type Browser

Jede in WinFS gespeicherte Datei (»Item« genannt, → *Kap. 4.4.4*) entspricht einem vorher definierten Schema, das von Anwendungen durch Vererbung erweitert werden kann:

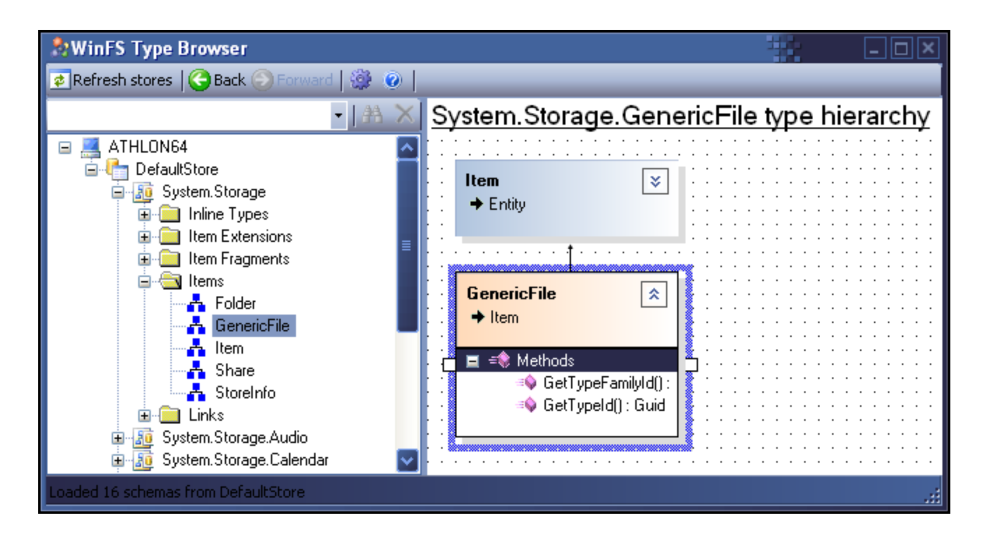


*Abb. 2.9-4: WinFS-Klasse **System.Storage.GenericFile***

Diese objektorientierte Typhierarchie erscheint als konsequente Weiterentwicklung von Rufus (→ *Kap. 2.3*) und vor allem Microsoft OLE (→ *Kap. 2.4*). Die Eigenschaften eines WinFS-Items umfassen auch vielseitige Metadaten, wie das Beispiel einer Videoaufzeichnung zeigt:

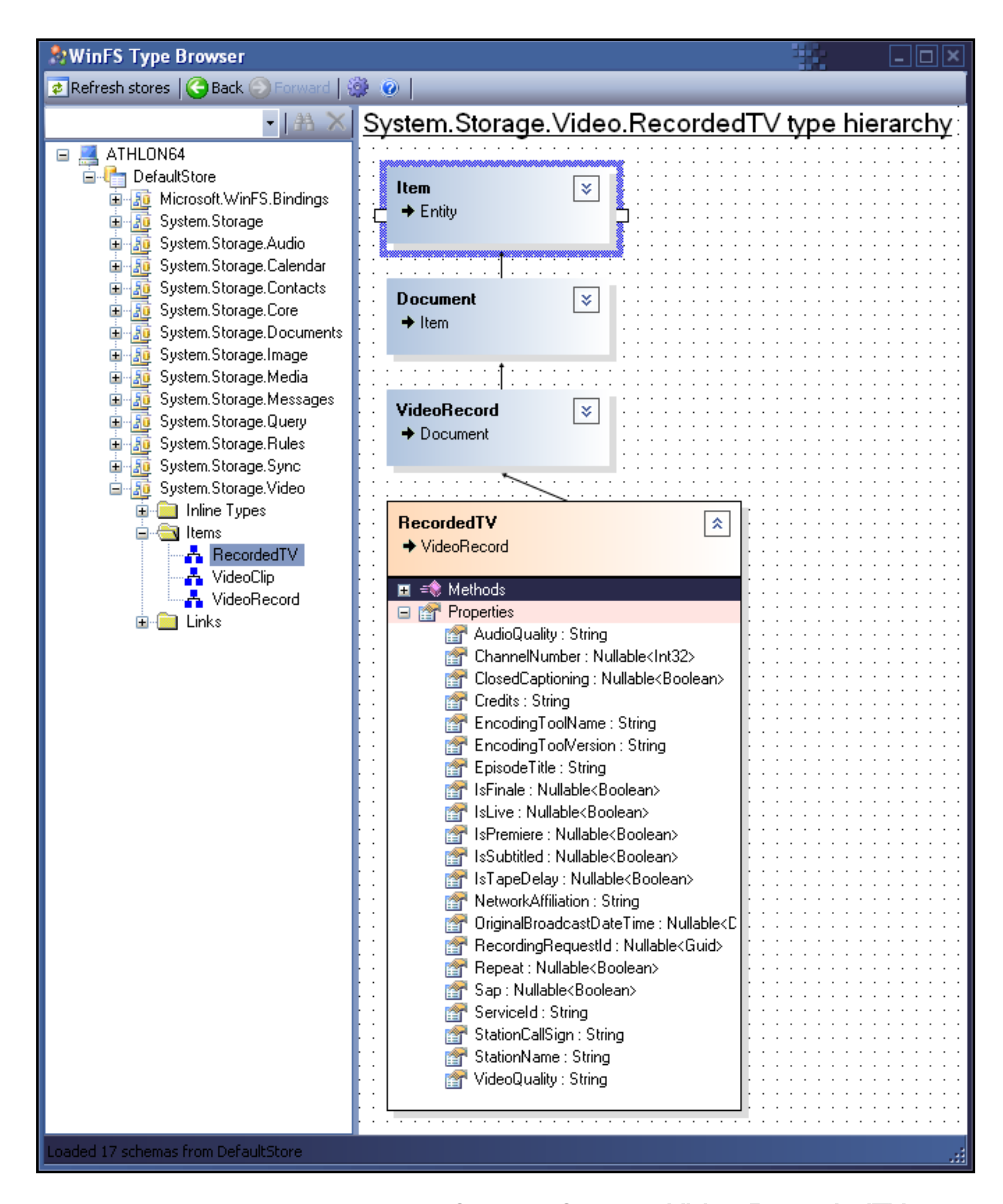


Abb. 2.9-5: WinFS-Klasse ***System.Storage.Video.RecordedTV***

2.9.2 StoreSpy

Der StoreSpy dient zur Betrachtung eines WinFS-Stores. Als Beispieldaten wurden etwa 400 JPEG-Bilder mit Exif-Metadaten **[EXI07]** und etwa 150 PDF-Dokumente in einen WinFS-Store kopiert. StoreSpy zeigt eine dreigeteilte Ansicht: links wird der Verzeichnisbaum des Stores angezeigt, in der Mitte ist eine Liste mit allen Dateien untergebracht, und rechts davon sind die Eigenschaften der ausgewählten Datei zu sehen:

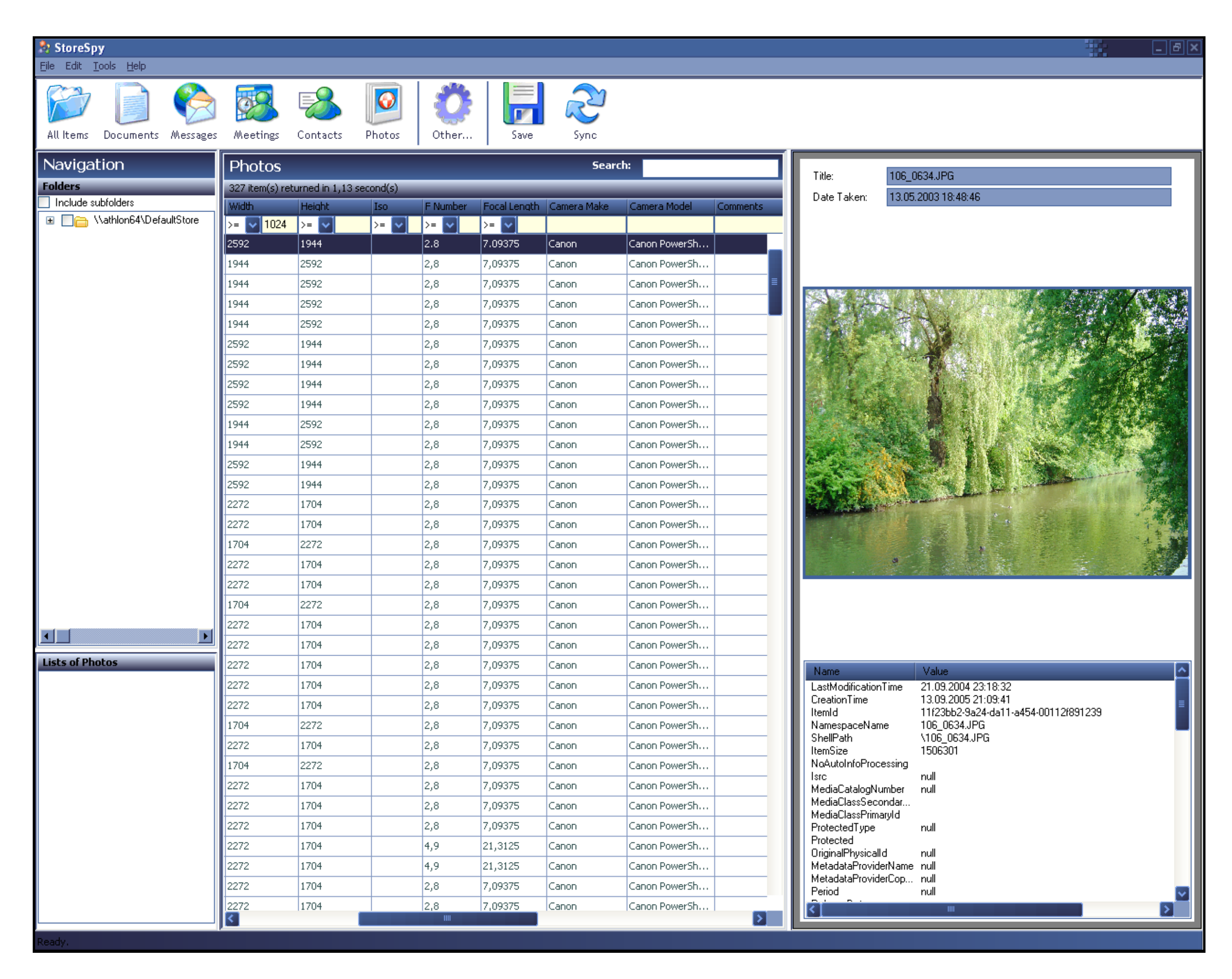


Abb. 2.9-6: StoreSpy

Die Liste der angezeigten Dateien wird dabei ohne Rücksicht auf mögliche Unterverzeichnisse erzeugt, und auch Metadaten werden angezeigt. Das Ergebnis kann nach Typgruppen (»Documents«, »Photos«, …) weiter gefiltert werden.

Das Besondere an WinFS liegt darin, dass die Dateiliste nicht mit herkömmlichen Funktionen des Betriebssystems gewonnen wird, sondern das Ergebnis einer Datenbank-Abfrage ist, die beliebige Attribute einschließen kann. In → *Abb. 2.9-6* werden nur Bilder angezeigt, deren Breite größer oder gleich 1024 Pixel ist. Diese Filterung der Dateien könnte beispielsweise durch das SQL-Statement `SELECT * FROM DefaultStore WHERE Width >= 1024` realisiert werden (tatsächlich verwendet WinFS eine proprietäre Abfragesprache namens »OPath« **[Mic08a]**).

Trotz eines leistungsstarken Rechners mit 64-Bit-CPU (Athlon 64 3200 +) und 1 GB RAM benötigte WinFS für die einfache Suche aller Items etwa 0,4 Sekunden, bei der oben gezeigten Filterung nach Bildgröße sogar 1,1 Sekunden. Weitere Versuche mit dem Microsoft SQL Server in → *Kap. 3.3.5.1* zeigen, dass die schlechte Performanz weniger WinFS als vielmehr einem ungeeigneten *DBMS* anzulasten ist.

2.10 Moderne Desktop-Suchmaschinen

Durch die große Beliebtheit des WWW haben diverse große Suchmaschinen-Betreiber wie Google basierend auf ihrer Suchtechnologie Produkte entwickelt, die Dateien im lokalen Dateisystem auffinden sollen. Ähnliche Produkte werden auch von zahlreichen anderen Herstellern angeboten. Im Gegensatz zu älterer Software wie Lotus Magellan (→ *Kap. 2.1*) haben moderne Desktop-Suchmaschinen eine große Verbreitung erlangt und somit eine hohe Relevanz. Sie entsprechen jedoch immer noch der ursprünglich mit Lotus Magellan eingeführten Architektur (→ *Abb. 2.1-1*), obwohl mittlerweile fast 20 Jahre vergangen sind. Viele dieser Lösungen stehen für diverse Betriebssysteme (Mac OS X, Linux, Windows) zum kostenfreien Download bereit oder sind sogar quelloffen:

- Aduna Autofocus
  http://aduna.biz/products/autofocus/
- Apple Spotlight (→ *2.10.2*)
  http://www.apple.com/macosx/features/spotlight/
- Ask Desktop
  http://sp.ask.com/docs/desktop/
- Beagle (→ *2.10.1*)
  http://beagle-project.org/
- Copernicus Desktop Search
  http://www.copernic.com/
- dtSearch Desktop
  http://www.dtsearch.com/
- diskMETA Pro
  http://www.diskmeta.com/
- Google Desktop (→ *2.10.3*)
  http://desktop.google.com/
- Microsoft MSN Toolbar
  http://toolbar.msn.com/
- Strigi
  http://strigi.sourceforge.net/
- xfriend
  http://www.x-friend.de/
- Yahoo Desktop (→ *2.10.4*)
  http://desktop.yahoo.com/

Da sich Desktop-Suchmaschinen sehr gleichen, werden in den folgenden Abschnitten nur die Besonderheiten einiger Suchmaschinen exemplarisch vorgestellt.

2.10.1 Linux Beagle

Mit dem Beagle-Projekt steht auch für Linux eine leistungsfähige Desktop-Suchmaschine auf Open-Source-Basis zur Verfügung. Beagle arbeitet wie üblich mit einem Index, der automatisch aktualisiert wird, wenn der Linux-Kernel so compiliert wurde, dass er die Funktion inotify unterstützt (Benachrichtigung, wenn Änderungen am Dateisystem vorgenommen wurden).

Beagle unterstützt nicht nur die Suche nach Dateien, sondern findet auch installierte Programme und greift auf Internet-Suchmaschinen zu. Es vereint damit verschiedene Suchfunktionen in einer einzigen Applikation **[Nov05]**:

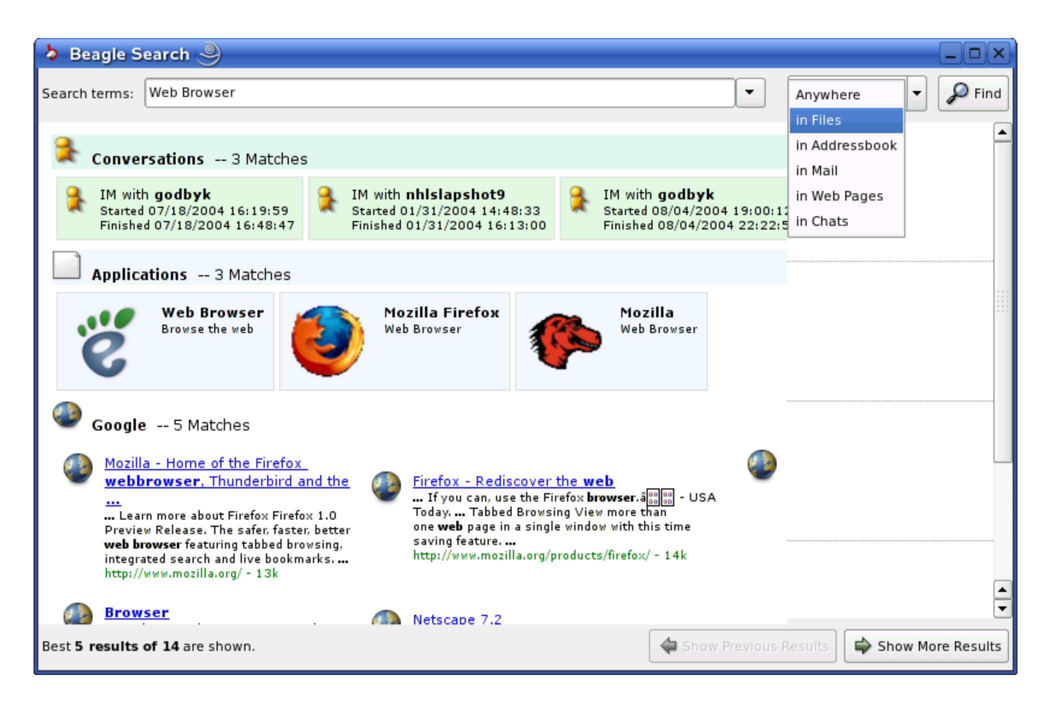


Abb. 2.10-1: *Linux Beagle* **[Nov05]**

2.10.2 Apple Spotlight

Die Hersteller von Betriebssystemen arbeiten ebenfalls daran, die Funktionalität von Desktop-Suchmaschinen in ihre Produkte zu integrieren. Das gilt z.B. für die Firma Apple, die Spotlight als Suchmaschine in Mac OS X ab Version 10.4 (Tiger) integriert. Frühere Versionen von Mac OS X gestatteten nur die Suche nach den klassischen Attributen des physikalischen Dateisystems:

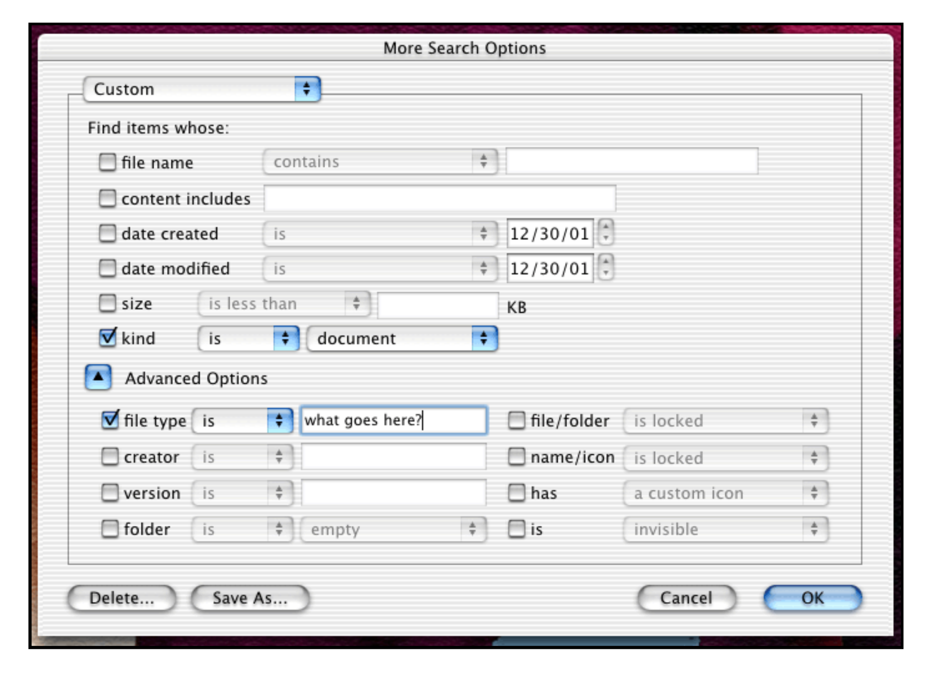


Abb. 2.10-2: Klassische Suchparameter unter Mac OS X

Spotlight fügt sich durch eine Eingabezeile für Suchbegriffe nahtlos in den *Finder* ein. Bei der Anzeige von Suchergebnissen stehen Kontrollelemente zum Eingrenzen der Suche zur Verfügung:

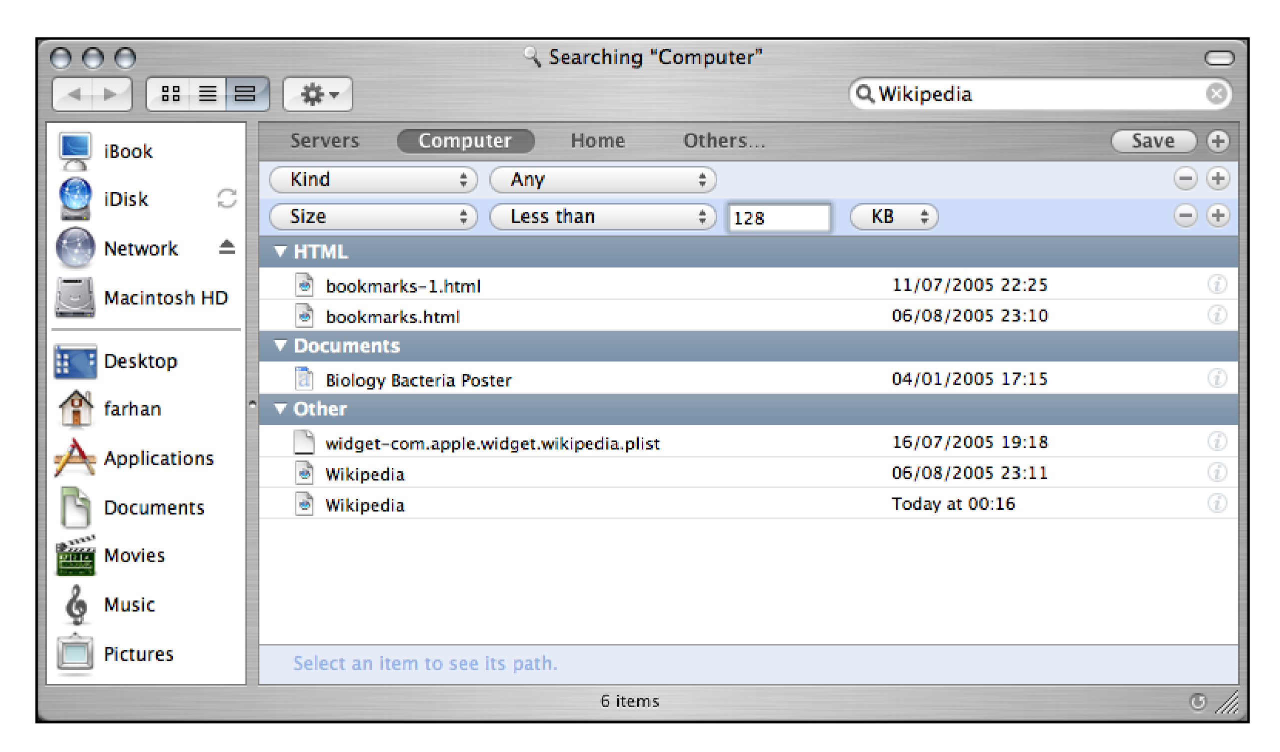


Abb. 2.10-3: Integration von Spotlight in den Finder **[App05]**

Genau wie die meisten anderen Systeme benötigt die Spotlight-Engine Zusatzmodule zur Bearbeitung diverser Dateiformate. Apple bietet zusätzliche Plug-Ins zum Download an, um Spotlight den Zugriff auf weitere Formate zu ermöglichen **[App08]**:

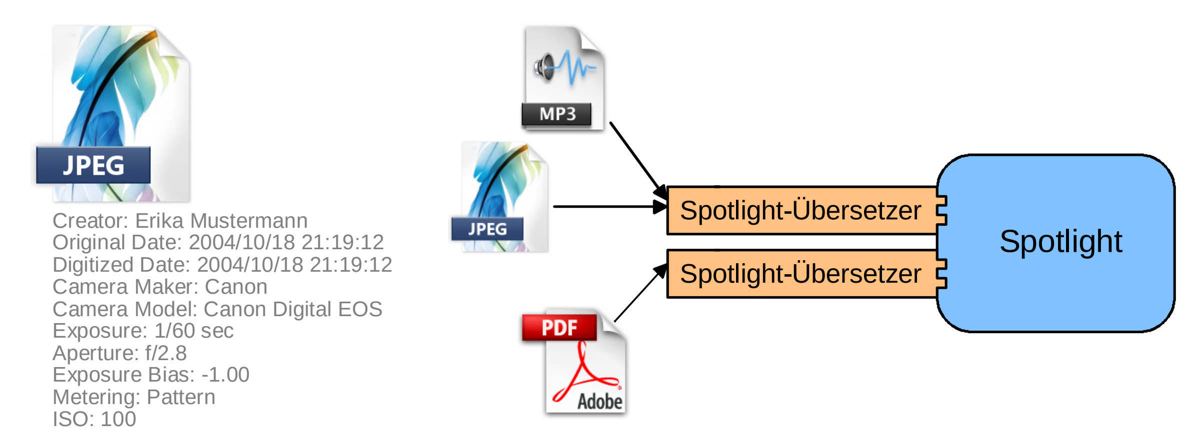


Abb. 2.10-4: Verarbeitung von Metadaten mit Apple Spotlight **[App05]**

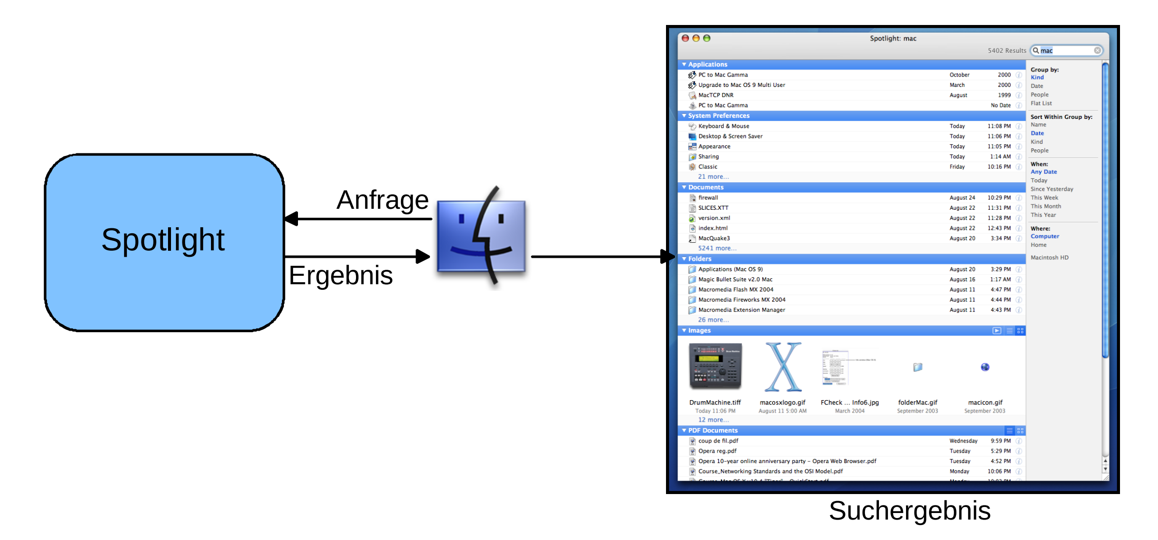


Abb. 2.10-5: Schematische Darstellung einer Suchanfrage mit Apple Spotlight **[App05]**

2.10.3 Google Desktop

Die Desktop-Suchmaschine »Google Desktop« wurde vom gleichnamigen Betreiber einer beliebten Internet-Suchmaschine entwickelt. Google Desktop integriert sich nahtlos in den Microsoft Internet Explorer und erweitert dort die Internet-Suchmaschine desselben Herstellers durch lokale Suchergebnisse und eine explizite Desktop-Suche.

Google Desktop nutzt zur Indexierung einen Wortindex, der lediglich Begriffe, nicht aber deren Herkunftsattribut indexiert. Das ist ein großer Nachteil, denn es kann nicht nach bestimmten Eigenschaften gesucht werden. Folgende Abbildung zeigt als Beispiel die Suche nach »1152«, beispielsweise um Bilder mit einer bestimmten Mindestauflösung zu finden:

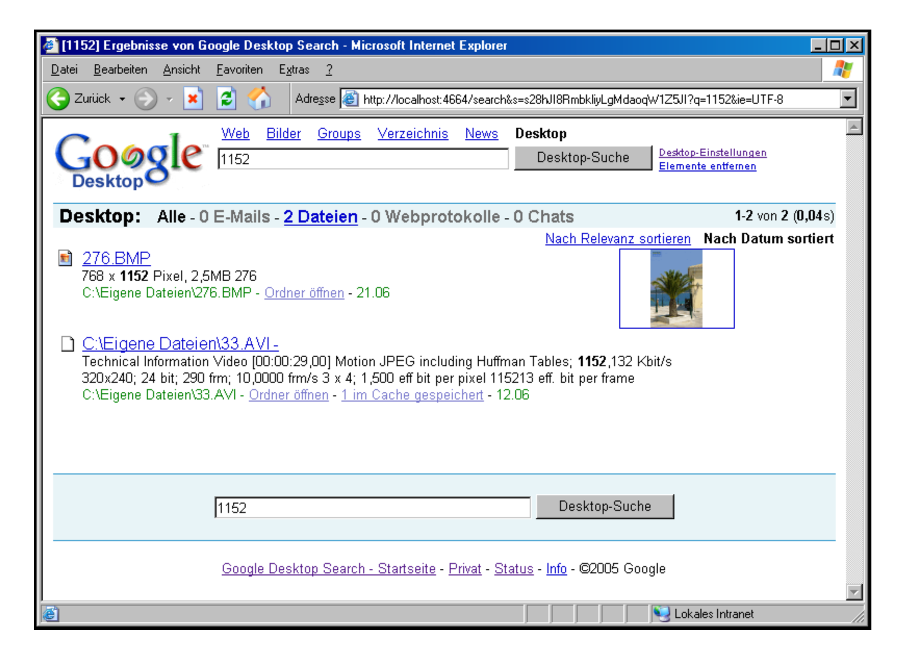


Abb. 2.10-6: Suche mit Google Desktop nach »1152«

Eine Suche nach dem Begriff »1152« findet nicht nur ein Bild mit dieser Höhe, sondern auch eine AVI-Videodatei mit einer Bitrate von 1152,132 kBit/s. Das Vorkommen in bestimmten Attributen, etwa nur der Bildhöhe, kann nicht abgefragt werden. Diese Schwäche des verwendeten Indexes ist besonders ärgerlich, weil die Internet-Suchmaschine durch Eingabe besonderer Parameter gesteuert werden kann: **filetype:pdf** liefert nur Dateien mit dieser Endung zurück, und **site:www.deskwork.de** findet nur Dokumente dieses WWW-Servers. Die Steuerung der Desktop-Suche durch Parameter wie **width:1024**, **minheight:1152** oder **artist:Anastacia** würde eine deutlich genauere Suche ermöglichen, was allerdings vom eingesetzten Wortindex nicht unterstützt wird.

2.10.4 Yahoo Desktop

Ein direktes Konkurrenzprodukt zu Google Desktop ist Yahoo Desktop, das ebenfalls von einem großen Betreiber einer WWW-Suchmaschine entwickelt wurde. Im Gegensatz zu Google Desktop benutzt Yahoo Desktop jedoch keinen WWW-Browser zur Darstellung der Suchergebnisse, sondern ist eine eigenständige Applikation.

An dieser Desktop-Suchmaschine fallen die Möglichkeiten zur Steuerung der Indexierung besonders positiv auf. Für jeden Teilbaum der Verzeichnishierarchie kann eingestellt werden, ob die dort enthaltenen Dateien vollständig, gar nicht oder nur mit ausgewählten Attributen indexiert werden sollen:

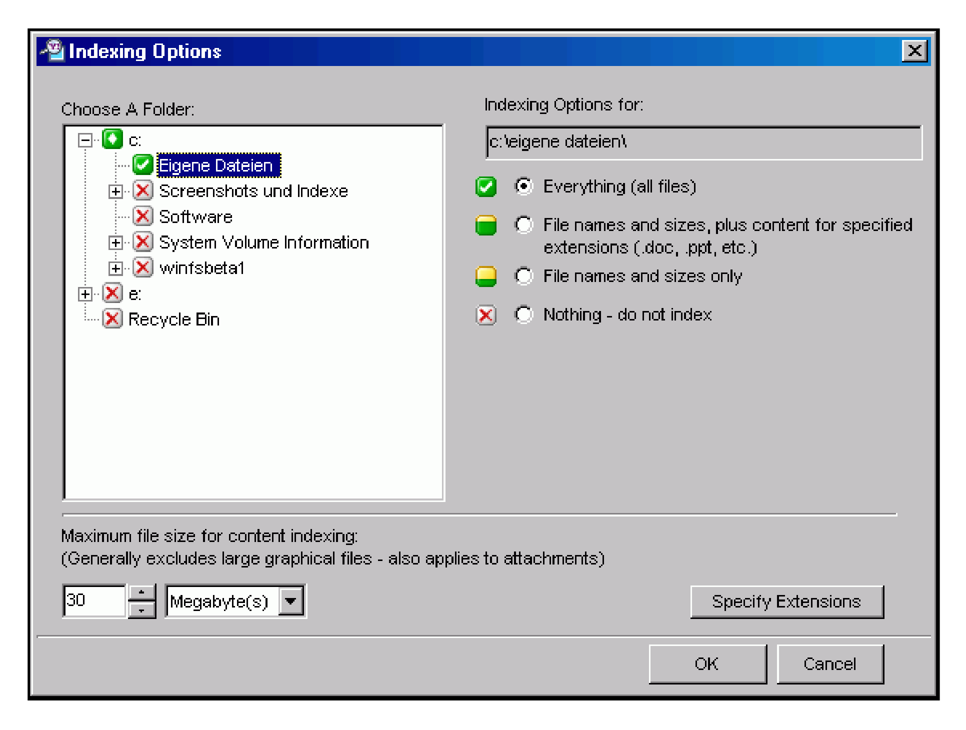


Abb. 2.10-7: Indexierungs-Optionen bei Yahoo Desktop

2.11 Fazit

Die Analyse verschiedener Systeme in diesem Kapitel dokumentiert die Anstrengungen, die zur Verbesserung physikalischer Dateisysteme unternommen wurden. Sie lassen sich, soweit zutreffend, bezüglich der Einbindung von Applikationen, den Verbesserungen an der Organisation des Dateisystems sowie anhand der eingesetzten Indexe bewerten.

Einige Lösungen versuchen, Dateien zusammen mit den zugehörigen Applikationen in eine objektorientierte Gesamtarchitektur einzubinden (→ *Kap. 2.3*), wozu als Grundlage alle Dateiformate mit ihren spezifischen Eigenschaften vom Betriebssystem registriert (→ *Kap. 2.4*) werden müssen. Da heterogene Dateiformate unterschiedliche Attribute und Operationen unterstützen, muss der Typ einer Datei sicher festgestellt werden können. Hierzu sind Dateiendungen nur bedingt geeignet, da sie vom Benutzer geändert werden können (→ *Abb. 4.4-4*). Ebenso ungeeignet sind die Klassifizierer des Rufus-Projekts (→ *Abb. 2.3.1*), da sie den Typ einer Datei nicht sicher ermitteln können.

Die Microsoft Indexerstellung (→ *Kap. 2.5*) zeigt als Bestandteil des Office-Pakets, dass ein Bedarf an besseren Organisationsformen für Dateien besteht. Eine Vielzahl von weiteren Systemen, wie z.B. Desktop-Suchmaschinen (→ *Kap. 2.10*), realisieren die Suche von Dateien über Verzeichnisgrenzen hinweg, ohne allerdings die eigentliche Organisation des Dateisystems zu verbessern. Diese Systeme sind also höchstens als Provisorium zu bezeichnen.

Die vorgestellten Lösungen, die eine Dateisuche ermöglichen, indexieren die Attribute des Dateisystems sowie Metadaten und ggf. Suchbegriffe aus dem Dateikörper. Der eingesetzte Index sollte dabei nicht nur das Vorkommen von bestimmten Attributwerten indexieren, sondern auch das Ursprungsattribut selbst, so dass gezielt nach Dateien mit bestimmten Eigenschaften gesucht werden kann. Ein Volltext-Index, wie er beispielsweise von Google Desktop eingesetzt wird, vermag dies nicht zu leisten, wodurch derartige Produkte nur eine eingeschränkte Funktionalität bieten können (→ *Kap. 2.10.3*).

Indexe, welche auch das Herkunftsattribut katalogisieren, werden im folgenden Kapitel → *3* vorgestellt. Aufgrund der Rahmenbedingungen von Dateisystemen bieten diese Indexe, die ursprünglich für andere Einsatzgebiete entwickelt wurden, nur eine eingeschränkte Leistung. Dies reduziert auch die Performanz des Gesamtsystems.

3 Existierende Indexe

Viele Lösungen, die in → *Kap. 2* vorgestellt wurden, verwenden einen Index zum Abspeichern von Metadaten und zur Bearbeitung von Suchanfragen. Indexe nehmen daher eine zentrale Rolle ein, die über die Performanz der Gesamtlösung entscheidet. Im folgenden Kapitel werden drei Ansätze für einen Index vorgestellt: die Lucene-Bibliothek (→ *Kap. 3.1*), die von vielen Desktop-Suchmaschinen eingesetzt wird, ein ins Dateisystem integrierter Index (→ *Kap. 3.2*) und multidimensionale Indexe (→ *Kap. 3.3*). Aufgrund der Integration von semantischen Dateisystemen und relationalen Datenbanken (→ *Kap. 4*) ist der letzte Ansatz besonders interessant, so dass die Ursachen für die dort auftretenden Performanz-Probleme ausführlich dargelegt werden. Die gewonnenen Erkenntnisse (→ *Kap. 3.4*) finden beim Master/Slave-Index (→ *Kap. 5*) Anwendung.

3.1 Lucene

Lucene ist eine Open-Source-Java-Bibliothek zum Erzeugen und Durchsuchen von Indexen; sie ist Teil der Apache Software Foundation. Mit Hilfe dieser plattformunabhängigen Bibliothek lassen sich in kurzer Zeit Volltextindexe für beliebige Inhalte erzeugen. Die Bibliothek setzt sich aus zwei Hauptbestandteilen zusammen: eine Komponente erzeugt den Index, wobei diesem beliebige typisierte Dokumente hinzugefügt werden. Eine »Query Engine« durchsucht den Index schließlich. Neben diesen grundlegenden Eigenschaften verfügt Lucene über eine reichhaltige Auswahl zusätzlicher Funktionen und Tools, welche durch die Open-Source-Community aktiv und umfangreich weiterentwickelt werden. Durch die hohe Performanz und Skalierbarkeit kann Lucene für beliebige Projektgrößen und Anforderungen eingesetzt werden **[Wik06]**, darunter auch die Desktop-Suchmaschinen Aduna AutoFocus, Beagle, Strigi und xfriend (→ *Kap. 2.10*) **[Luc07]**.

3.1.1 Compound File System

Lucene speichert den Index im sog. »Compound File System« ab, was im übertragenen Sinn ein begrenztes Gelände meint. Dieses »Gelände« besteht aus den Dateien **deletable**, **SEGMENTS** und dem eigentlichen Index mit der Endung **CFS**:

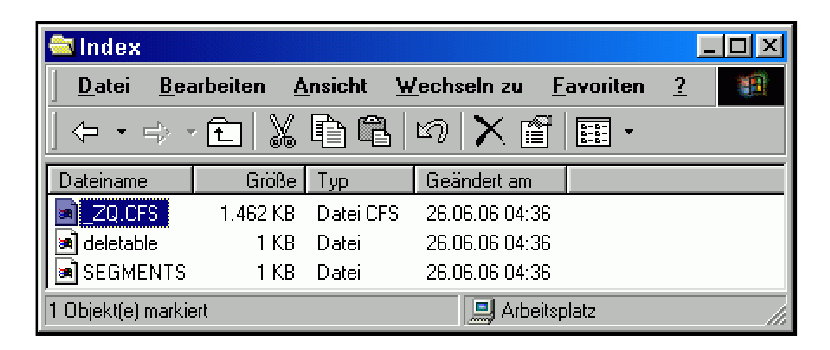
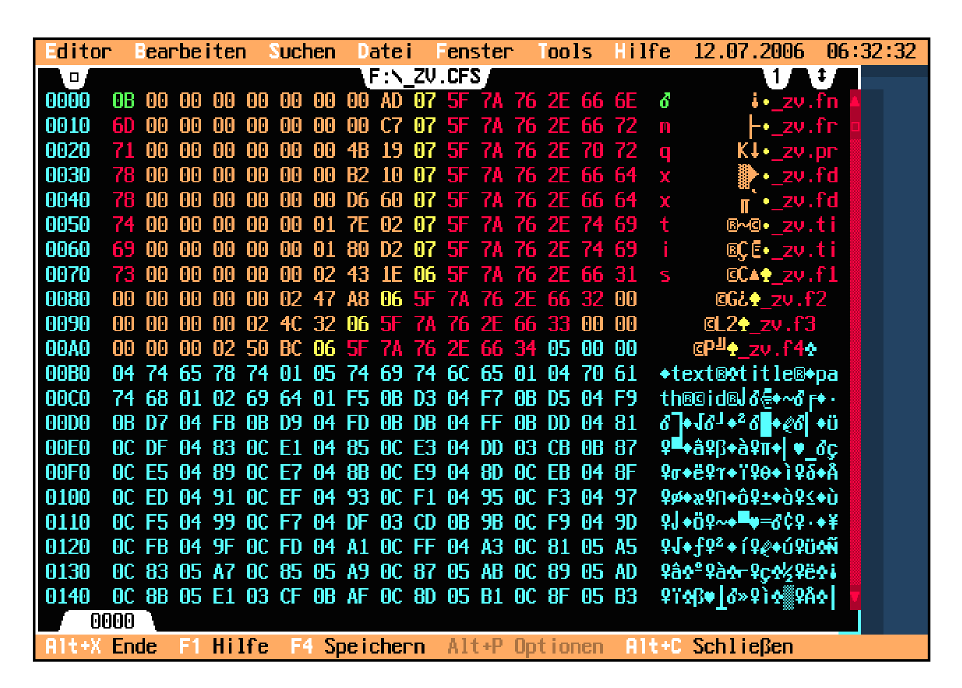


Abb. 3.1-1: Von xfriend angelegter Lucene-Index

Die **CFS**-Datei enthält eine Sammlung verschiedener Einzeldateien, die unkomprimiert vorliegen. Der Header enthält eine Aufstellung aller enthaltenen Unterdateien mit Anzahl, Offset und Name:



Grün hervorgehoben: Anzahl der Dateien im Compound File System
Orange hervorgehoben: Dateioffset als int64
Gelb hervorgehoben: Längenbyte vor Strings
Rot hervorgehoben: Dateinamen

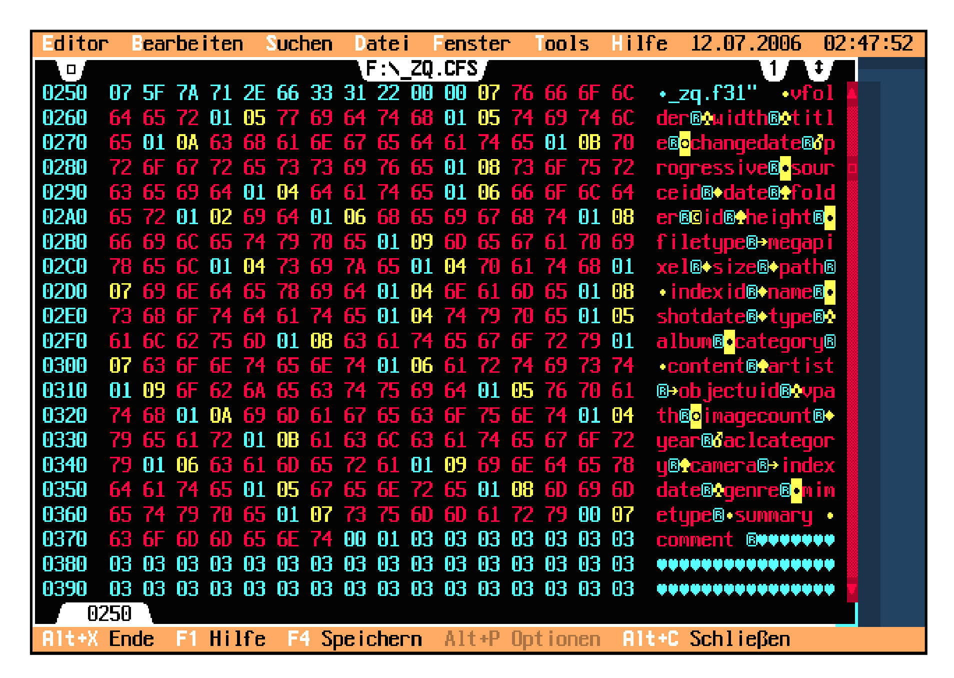
Abb. 3.1-2: Dateiheader eines Compound File System von Aduna Autofocus (→ Kap. 2.10*)*

Die im Compound File System enthaltenen Einzeldateien erfüllen folgende Funktionen **[Luc06]**:

	Funktion
.fnm	**Feldnamen** (→ *Abb. 3.1-4*) Attribute werden nach der Position in der Namensliste durchnummeriert.
.frq	**Worthäufigkeit** (frequency) Enthält zu jedem Wort die Dateien, in denen es vorkommt (einschl. Häufigkeit)
.prx	**Nähe zu anderen Worten** (proximity) Enthält die Position eines Wortes innerhalb des Dokuments
.fdx	**Feldindex** Enthält für alle Dokumente Zeiger auf die eigentlichen Metadaten, falls vorhanden
.fdt	**Felddaten** Enthält die eigentlichen Metadaten
.tis	**Wörterbuch** (term infos) Enthält alle Worte, die in den indexierten Dateien vorkommen

Abb. 3.1-3: Funktionen von Unterdateien im Compound File System **[Luc06]**

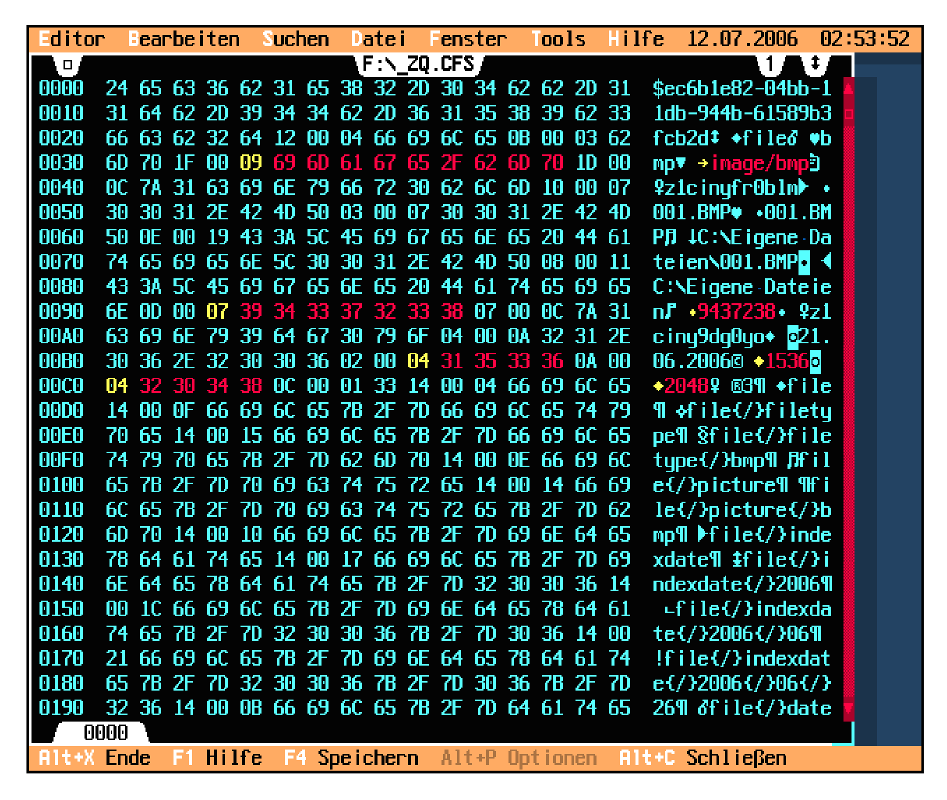
Damit ist offensichtlich, dass Lucene einen Index in Form von invertierten Listen **[Kol07b]** aufbaut. Am Anfang der Index-Datei befindet sich eine Aufstellung aller vorkommenden Attribute. Da xfriend vor allem auf Multimedia-Dateiformate spezialisiert ist, finden sich im Beispielindex u.a. geläufige Metadaten wieder:



Gelb hervorgehoben: Längenbyte vor Strings
Rot hervorgehoben: Attributnamen

Abb. 3.1-4: Im xfriend-Index enthaltene Attributnamen

Weitere Informationen zu den einzelnen Attributen folgen in der entsprechenden Unterdatei (Endung .fdt) direkt aufeinander. Eine Trennung ist nicht erforderlich, da der Feldindex (Endung .fdx) für jede Datei Zeiger auf die entsprechenden Daten verwaltet:



Gelb hervorgehoben: Längenbyte vor Zeichenketten
Rot hervorgehoben: MIME-Typ, Dateilänge und Auflösung

Abb. 3.1-5: Felddaten einer Bilddatei

3.1.2 Performanz

Invertierte Listen, wie sie Lucene verwaltet, sind problematisch für den Einsatz in Desktop-Suchmaschinen, da Änderungen am Dateisystem sehr umfangreiche Aktualisierungen des Indexes implizieren **[Saa05]**. Besonders zeitaufwändig ist die komplette Neuindexierung des Dateisystems. Beagle implementiert den Indexer daher als *Leerlauf-Prozess*, wodurch sich das System zwar nicht verlangsamt, eine vollständige Indexierung aber besonders viel Zeit beansprucht:

> Obwohl es sich noch um Alphaware handelt, macht das Tool einen soliden Eindruck. Einzig die Installation und die erste Indexerstellung sind etwas problematisch. … Die Indexerstellung hat bei mir (50 GB Home circa 48h gedauert – wobei die Systembelastung nicht weiter ins Gewicht fiel – ich konnte normal weiterarbeiten)

Abb. 3.1-6: Private WWW-Seite unter **[Lin05]**

Dieses Verhalten ist jedoch tatsächlich problematisch, denn nach der Installation sollte ein Programmpaket möglichst sofort einsatzbereit sein, jedenfalls nicht erst nach zwei Tagen Laufzeit. Diese Nachteile sind jedoch nicht Lucene anzulasten, da die Bibliothek als universeller Textindex für eine Vielzahl von Einsatzzwecken entwickelt wurde. Vielmehr sind es die Hersteller von Desktop-Suchmaschinen, die für ihre Projekte geeignetere Alternativen zu Lucene einsetzen sollten – etwa einen Master/Slave-Index (→ *5*).

Darüber hinaus fällt auf, dass alle Attributwerte einer Datei als Zeichenkette gespeichert werden: die Auflösung der Bilddatei (→ *Abb. 3.1-5*) ist als **'1536'** und **'2048'** enthalten und nicht als Binärzahl. Dies ist effizient für die Suche durch Schlagworte, wie sie Internet-Suchmaschinen verwenden, erschwert allerdings Vergleiche mit den Operatoren »größer« oder »kleiner«.

3.2 BeFS

Die Funktionsweise von BeFS ähnelt der anderer Unix-Dateisysteme wie beispielsweise ext2, erweitert um einige Aspekte eines semantischen Dateisystems (→ *Kap. 2.7.2*). Hier wird vor allem die Verwaltung der Metadaten (→ *Kap. 2.7.3*) diskutiert. Weiterführende Informationen hierzu sind in **[Gia99]** zu finden.

BeFS speichert Attribute als Paar von Schlüsselwort und Wert ab, in Analogie zu relationalen Datenbanken. Die Fähigkeit, beliebig viele Attribute anzulegen, gestattet das einfache Abspeichern von Metadaten, die auch automatisch indexiert werden können. BeFS benutzt einen *I-Node*, um wichtige Informationen über eine Datei abzuspeichern. Die Struktur heißt **bfs_inode** und hat folgenden Aufbau **[Gia99]**:

```
typedef struct bfs_inode
{
  int32         magic1;
  inode_addr    inode_num;
  int32         uid;
  int32         gid;
  int32         mode;
  int32         flags;
  bigtime_t     create_time;
  bigtime_t     last_modified_time;
  inode_addr    parent;
  inode_addr    attributes;
  unit32        type;
  int32         inode_size;
  binode_etc    *etc;
  data_stream   data;
  int32         pad[4];
  int32         small_data[1];
} bfs_inode;
```

Besonders auffällig sind die Einträge vom Typ **inode_addr**; sie verweisen auf andere I-Nodes. Dabei ist **inode_num** ein Verweis auf sich selbst, und **parent** zeigt auf den I-Node des Ordners, in dem sich die Datei bzw. das Verzeichnis befindet. Besonders wichtig für die Verwaltung von Metadaten ist die Variable **attributes**, welche auf den I-Node eines versteckten Ordners zeigt, der alle Attribute speichert. Dieses Verzeichnis befindet sich außerhalb des normalen Verzeichnisbaums und wird mit speziellen API-Funktionen angesprochen. Der Vorteil dieser Vorgehensweise liegt darin, dass der Code der normalen Verzeichnisverwaltung für Attribute wiederverwendet werden kann. In einem ersten Entwurf von BeOS bestand eine Datei daher aus folgenden Komponenten **[Gia99]**:

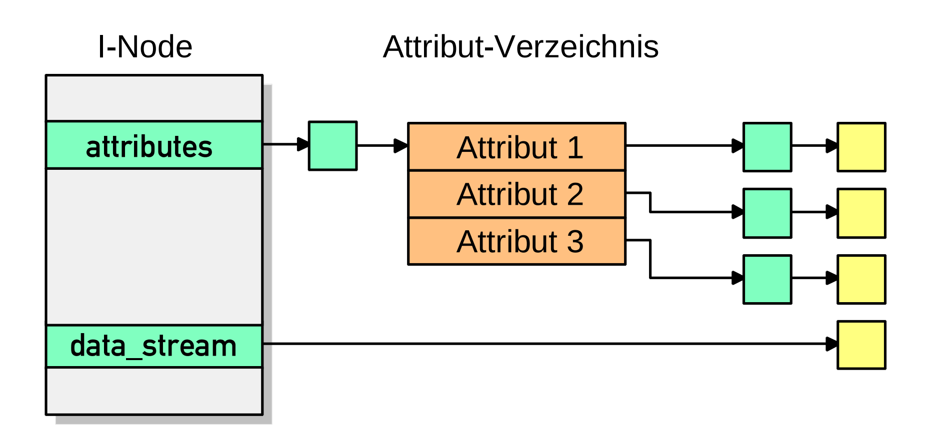


Abb. 3.2-1: Datei mit Attributen **[Gia99]**

Diese Implementierung erscheint zunächst sehr elegant, sie hat allerdings gravierende Nachteile. Für den Zugriff auf ein bestimmtes Attribut muss ausgehend vom I-Node der Datei der I-Node des Attribut-Verzeichnisses eingelesen werden. Die dortige Variable **data_stream** verweist auf die Position der Verzeichnisdaten auf der Festplatte – bei BeFS ein B^+-Baum **[Saa05]**, der die Nummern der I-Nodes der einzelnen Attribute enthält. Ein solcher I-Node enthält wiederum eine Variable **data_steam**, die dann die Position der eigentlichen Attribute auf der Festplatte speichert, die zusammen den B^+-Baum bilden. Der Zugriff auf ein Attribut ist also vierfach-indirekt **[Gia99]**.

Da BeOS ein rein grafisches Betriebssystem ist, sind neben den eigentlichen Metadaten aus den Dateien zugleich viele von der GUI erzeugte Attribute zu speichern, etwa das anzuzeigende Symbol oder die Position im Fenster des Verzeichnisses. Wird nun vom Benutzer ein Ordner geöffnet, so müssen diese Angaben von allen Dateien im Ordner gelesen werden. Da auf die eigentlichen Daten nur vierfach-indirekt zugegriffen werden kann, impliziert dieser Vorgang einen erheblichen Zeitaufwand. Ergänzend kommt hinzu, dass die meisten Attribute nur wenige Byte groß sind, die Architektur mit zusätzlichen I-Nodes also sehr viel Speicherplatz verschwendet **[Gia99]**.

Da ein I-Node nur etwas mehr als 200 Byte belegt, bleiben bei der minimalen Clustergröße von 1024 Byte (2 Sektoren) knapp 800 Byte im I-Node ungenutzt. Daher wurde die Architektur so abgeändert, dass kleine Attribute direkt innerhalb des I-Nodes gespeichert werden. Ein Attribut-Verzeichnis wird nur bei größeren Attributen benötigt, zum Beispiel für das »To«-Feld einer EMail mit vielen Empfängern **[Gia99]**.

3.2.1 Indexierung

BeOS legt für viele Attribute direkt im Dateisystem einen Index an, um eine schnelle Suche zu unterstützen. Jeder Index verwaltet seine Einträge in einem B^+-Baum, so dass der Index als unsichtbares Verzeichnis auf der Festplatte abgelegt und der entsprechende Programmcode wieder verwendet werden kann **[Gia99]**.

Das Dateisystem unterstützt Indexe für die Datentypen **String** (bis 255 Zeichen), **Int** (32 und 64 Bit), **Float** und **Double** als Schlüssel. Das Anlegen eines Index bzgl. eines bestimmten Attributs wird über eine API-Funktion von einer Applikation veranlasst. Das Dateisystem indexiert zudem von sich aus Dateiname, -größe und das Datum der letzten Änderung einer jeden Datei **[Gia99]**.

3.2.2 Performanz

Die oben dargestellte Realisierung eines Metadaten-Indexes durch B^+-Bäume **[Saa05]** ist aus Implementierungssicht zwar elegant, da bestehende Datenstrukturen und zugehöriger Programmcode benutzt werden können, allerdings merken die BeOS-Entwickler selbst an, dass Bäume beim Einsatz in Dateisystemen Nachteile haben:

> Early versions of BFS did in fact index the creation time of files, but we deemed this index not to be worth the performance penalty it cost. By eliminating the creation time index, the file system received roughly a 20% speed boost in a file create and delete benchmark.

Abb. 3.2-2: Performanz-Einbußen durch Indexierung der Dateizeit **[Gia99]**

Es war daher nur konsequent, »kleine« Attribute, insbesondere solche, nach denen nicht gesucht wird (wie die Position des Icons im Fenster) direkt im I-Node der Datei unterzubringen. Dies ist je-

doch nur für eine begrenzte Anzahl von Attributen möglich und sinnvoll. Alle Attribute, nach denen gesucht werden soll oder die nicht in den I-Node passen, müssen weiterhin in einem B^+-Baum untergebracht werden, auf dessen Blätter von jedem I-Node aus mittels Attribut-Verzeichnis verwiesen wird.

Dies führt zu gravierenden Leistungseinbußen, da bei zahlreichen Attributen eine entsprechend große Anzahl von Baumindexen notwendig wird, die beim Anlegen, Verändern und Löschen einer Datei in ihrer Gesamtheit aktualisiert werden müssen. Die Entwickler von BeOS haben versucht diesen Effekt zu minimieren, indem nicht alle Attribute indexiert werden (→ *Abb. 3.2-2*). Zusätzlich degeneriert die Zugriffsgeschwindigkeit von Baumstrukturen, wenn sie auf mechanischen Festplatten gespeichert werden, da hier der Zugriff auf verteilt gespeicherte Datenblöcke besonders zeitaufwändig ist (→ *Kap. 3.3.3*). Dieser Effekt hat Benoit Schillings, einer der Entwickler von BeFS, in einem Interview zu folgender Aussage veranlasst:

> And I didn't use B-Trees. I thought B-Trees were evil, and still do. Reading them is slow and super-expensive. … Reading big blocks is actually more efficient than using B-Trees on modern hardware …

Abb. 3.2-3: Windows on a database – sliced and diced by BeOS vets **[Reg02]**

3.3 Multidimensionale Indexierung

Ein zentrales Problem relationaler Datenbanksysteme ist das multidimensionale Indexieren, also das Indexieren mehrerer Attribute in einer einzigen Datenstruktur. Dieses Problem betrifft auch Dateisysteme, da Metadaten aus zahlreichen Attributen bestehen: bei vollständiger Implementierung des Exif- **[EXI07]** und ID3-Standards **[Nil05]** wird ein Datenraum mit mehr als hundert Dimensionen erzeugt, der damit nicht nur multi-, sondern sogar hochdimensional ist. Auf derartig beschaffene Daten und zugehörigen Indexen können im Sinne der Datenbanktechnik verschiedene Klassen von Operationen (in der Regel Suchanfragen) angewandt werden:

	Full Match	**Partial Match**
Exakt	Alle Objekte mit exakten Werten in allen Attributen	Alle Objekte mit exakten Werten in **wenigen** spezifizierten Attributen
Nachbarschaft	Alle Objekte, die höchstens eine Entfernung ε vom spezifizierten Punkt haben	

Abb. 3.3-1: Klassifizierung von multidimensionalen Operationen

In der folgenden Abbildung werden existierende Indexstrukturen für multi- und hochdimensionale Datenräume nach zwei wichtigen Kriterien klassifiziert, nämlich der Eignung für bestimmte Operationen aus → *Abb. 3.3-1* und dem Typ der Suche (Baum oder sequenzieller Scan):

	Partial Match Operationen effizient unterstützt	Partial Match Operationen nicht effizient möglich
Baum ↑	*d* B-, B$^+$-Bäume	A-Baum iDistance kd-, kdB-Baum R-, R$^+$, R*-Baum SS-Baum
		X-Baum
Seq. ↓	Sequenzieller Scan	Gridfiles Hashing VA-Datei

d: Anzahl der Dimensionen des Datenraums

Abb. 3.3-2: Klassifizierung von multidimensionalen Indexstrukturen **[Sak00]**

In **[Saa05]** werden einige Techniken zur multi- bzw. hochdimensionalen Indexierung vorgestellt, darunter auch das multidimensionale Hashen, kd- bzw. kdB-Bäume **[Ben75]** sowie R-Bäume **[Gut84]**. Insbesondere auf kd- und R-Bäumen basieren viele Derivate, u.a. R$^+$- **[Sel87]**, R*- **[Bec90]**, X- **[Ber96]**, SS- **[Whi96]** und A-Bäume **[Sak00]**.

3.3.1 Der »Fluch der Dimensionen«

Die meisten der oben erwähnten Datenstrukturen zeigen bei hohen Dimensionszahlen ($d > 10$) gravierende Geschwindigkeitsverluste. Das gilt insbesondere für Indexe, die auf einer Partitionierung des Datenraums Ω oder Objektraums ω basieren, also u.a. auch für kd-Bäume, R-Bäume und ihre Derivate. Die mathematische Grundlage für diesen Effekt liegt in der exponentiellen Zunahme des Raumvolumens beim Hinzufügen von Dimensionen. **[Ber96]** untersucht, wie sich dieser Effekt auf R-Bäume auswirkt. Bei großen Dimensionszahlen weisen die umschreibenden Rechtecke einen großen Überlappungsgrad auf, der sogar schon bei $d = 2$ beobachtet werden kann:

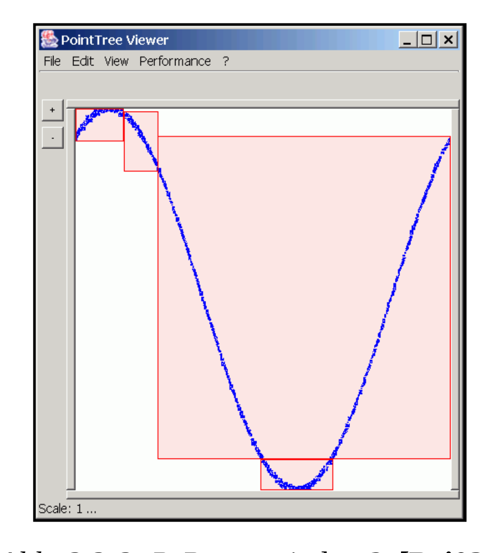


Abb. 3.3-3: R-Baum mit d = 2 **[Bri03]**

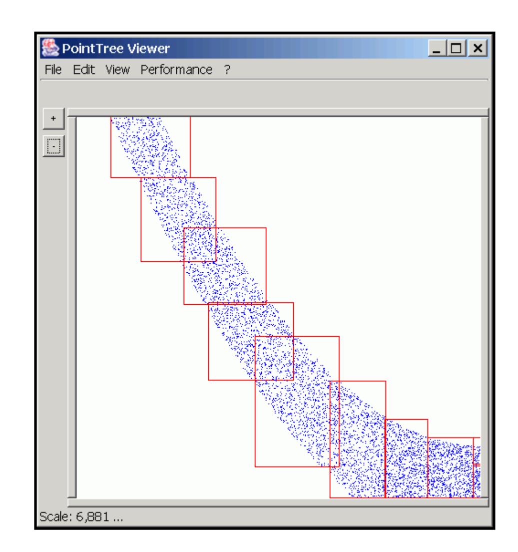


Abb. 3.3-4: R-Baum (Vergrößerung) **[Bri03]**

Bereits bei zehn Dimensionen wird eine Überlappung von 100% erreicht, was R-Bäume für hochdimensionale Anwendungen unbrauchbar macht. Das ist ein Symptom eines grundlegenden Effekts, der »Fluch der Dimensionen« genannt wird. Von diesem Effekt sind alle Nachbarschafts-Operationen betroffen, bei denen sowohl der Datenraum als auch die Suchanfrage hochdimensional sind.

Für exakte Full Match Operationen hingegen eignen sich Gridfiles **[Saa05]**, die Suchanfragen in konstanter Zeit beantworten. Full Match Operationen spielen für Dateien jedoch keine wichtige Rolle, da der Benutzer nicht alle Attributwerte einer Datei spezifizieren kann, und eine gegebene Datei in der Regel auch nicht alle Attribute besitzt (→ *Kap. 3.3.4*), um einen genauen Referenzpunkt im globalen Datenraum Ω zu definieren. Dennoch werden im Folgenden Indexstrukturen für diese Operationen untersucht, da auf diese Weise wertvolle Erkenntnisse für die Indexierung von Dateien gewonnen werden.

3.3.1.1 Triviale Extreme

Die Nachbarschaftssuche in einem d-dimensionalen Datenraum beinhaltet das Spezifizieren eines Punktes p und die Suche nach dem Datenobjekt, das den geringsten Abstand zu p aufweist. Eine Erweiterung, die k-Nachbarschaftssuche, soll die k nächstgelegenen Objekte liefern **[Bor99]**.

Für diese Probleme existieren zwei triviale Algorithmen, die entweder die Suchzeit oder den Speicherbedarf minimieren. Zum einen kann für alle Objekte der Abstand zu p ausgerechnet werden, wobei das Objekt mit dem kleinsten Abstand gespeichert und zurückgeliefert wird. Die Ausführungszeit beträgt $O(n)$, der zusätzliche Speicherbedarf ist konstant. Der andere Extremfall existiert nur, wenn eine endliche Anzahl von Suchanfragen möglich ist: dann kann das Ergebnis für jede denkbare Anfrage im Voraus berechnet und gespeichert werden. Suchanfragen können dann in $O(d)$ beantwortet werden, allerdings wird dies mit einem exponentiellen Speicherbedarf erkauft **[Bor99]**.

Das eigentliche Problem der Nachbarschaftssuche besteht nun darin, gleichzeitig die Laufzeit und den Speicherbedarf zu minimieren. Für sinnvolle Umgebungen (z.B. euklidischer Datenraum) und beliebige n und d scheint kein Algorithmus zu existieren, der sowohl die Laufzeit auf $poly(d)$ und den Speicherbedarf auf $poly(nd)$ begrenzt. Dieses Phänomen ist als »Fluch der Dimensionen« bekannt **[Bor99]**.

3.3.1.2 Partitionierung des Daten- bzw. Objektraums

Durch die Wichtigkeit hochdimensionaler Datenräume für praktische Anwendungen wurde in der Vergangenheit intensiv nach Datenstrukturen und Algorithmen geforscht, die die Vorteile der beiden Extremlösungen vereinen sollen. Dabei fällt auf, dass vor allem das sequenzielle Scannen aller Objekte bei geschickter Implementierung eine überraschend gute Performanz bietet und unter bestimmten Voraussetzungen sogar optimal für große d ist **[Web98]**.

Indexstrukturen, die den Daten- bzw. Objektraum partitionieren, degenerieren mit zunehmender Dimensionszahl, so dass die Partitionen zunehmend große Volumina enthalten und sich daher stark überlappen. Dadurch müssen sehr viele Objekte betrachtet werden, so dass die Partitionen letztlich doch sequenziell gescannt werden – **zusätzlich** zum Aufwand, der durch die Partitionierung verursacht wird (etwa das Traversieren eines Baums). **[Ber96]** schlägt daher einen modifizierten R-Baum vor, den X-Baum (»extended node tree«), bei dem derartig degenerierte Baumknoten absichtlich durch lineare Listen (»supernodes«) ersetzt und dann sequenziell durchsucht werden:

> The X-tree may be seen as a hybrid of a linear array-like and a hierarchical R-tree-like directory. … It is also reasonable, that for very high dimensionality a linear organization of the directory is more efficient. The reason is that due to the high overlap, most of the directory if not the whole directory has to be searched anyway.

Abb. 3.3-5: Beschreibung des X-Baums **[Ber96]**

[Web98] beweist schließlich, dass dieser Sachverhalt nicht nur für R-Bäume, sondern unter Annahme der Gleichverteilung für jeden denkbaren multidimensionalen Index gilt, der den Daten- oder Objektraum partitioniert. Dieser Beweis wird nun kurz skizziert.

Alle Attributwerte werden zunächst auf das Intervall [0,1] skaliert, so dass der Datenraum auf einen Hyperwürfel mit der Kantenlänge 1 normiert wird: $\Omega = [0,1]^d$. Innerhalb des Datenraums soll ein beliebiger Algorithmus Elemente zu »Bereichen« zusammenfassen, entweder durch Unterteilen des Datenraums Ω ohne Rücksicht auf die tatsächlich enthaltenen Daten (z.B. kd-Baum) oder durch Clustern der tatsächlich vorhandenden Objekte, also Partitionieren des Objektraums ω (z.B. R-Baum). Die entstehenden Bereiche haben drei grundlegende Eigenschaften **[Web98]**:

1. Jeder Bereich besitzt eine minimal umgebende Form (»minimum bounding region«).
2. Jede *mbr* ist konvex.
3. Jeder Bereich enthält mindestens zwei Elemente. Ohne diese Bedingung würde das Unterteilungsproblem nur auf eine andere Knotenebene der Baumstruktur verschoben.

Die Wahrscheinlichkeit, dass bei einer Nachbarschaftssuche ein bestimmter Cluster C_i von l Clustern insgesamt betrachtet werden muss (und somit die entsprechenden Sektoren mit den Attributen geladen werden), ist proportional zum Volumen, den C_i im Suchraum einnimmt:

$$VM(x) \equiv Vol\left(MSum\left(x, E[nn^{dist}]\right) \cap \Omega\right)$$
$$p_{visit}^{avg} = \frac{1}{l}\sum_{i=1}^{l} VM\left(mbr\left(C_i\right)\right) \tag{3.1}$$

$VM(x)$ bezeichnet das Volumen von Cluster x, das durch die Minkowski-Summe (Funktion *MSum*) um die erwartete Entfernung $E[nn^{dist}]$ des nächsten Nachbarn von x vergrößert wird, begrenzt vom Datenraum Ω. Das Volumen eines Bereichs C_i kann nach unten begrenzt werden, d.h. es kann ein

Mindestvolumen angegeben werden: da die *mbr* eines jeden Bereichs konvex ist, verlässt eine Hilfslinie zwischen zwei beliebigen Elementen x_i und y_i eines Bereichs C_i die *mbr* nicht, so dass gilt **[Web98]**:

$$x_i, y_i \in C_i$$
$$VM(mbr(C_i)) \geq VM(line(x_i, y_i)) \tag{3.2}$$

Damit kann auch die durchschnittliche Wahrscheinlichkeit für den Zugriff auf C_i nach unten begrenzt werden. Die von Weber durchgeführten und in **[Web98]** beschriebenen Simulationen zeigen nun, dass bei 10^6 Elementen im Suchraum Ω ab 1500 Dimensionen nahezu alle Cluster betrachtet werden müssen. Ausgehend davon, dass ein sequenzielles Lesen des Indexes im Vergleich zu wahlfreiem Zugriff auf moderner Hardware nur einen Bruchteil der Zeit benötigt (→ *Kap. 3.3.3.1*), ergeben sich folgende Schlussfolgerungen **[Web98]**:

1. Für jeden Partitions- oder Clusterindex existiert eine Dimensionszahl *d*, ab der ein sequenzieller Suchlauf schneller ist. In der Praxis ist *d* dabei deutlich kleiner als 610.
2. Die Zeitkomplexität jedes Partitions- oder Clusterindex konvergiert bei großen *d* gegen $O(n)$.
3. Für jeden multidimensionalen Partitions- oder Clusterindex gibt es eine Dimensionszahl *d*, bei deren Überschreiten alle Sektoren des Index gelesen werden müssen.

Die Überlegenheit des sequenziellen Scannens bei Gleichverteilung wird von **[Jag05]** experimentell bestätigt. Als Konsequenz aus den theoretischen Überlegungen wird in **[Web98]** eine Indexstuktur namens VA-Datei entwickelt, die approximierte Attributwerte abspeichert und sequenziell scannt. Für alle (im Idealfall wenige) Objekte, bei denen die approximierten Werte in der Nachbarschaft des Referenzpunktes liegen, wird dann der tatsächliche Abstand zu *p* berechnet.

Liegt keine Gleichverteilung vor, so existieren Indexstrukturen wie iDistance **[Jag05]**, die bei Nachbarschaftssuchen effizienter sind als VA-Dateien. Der Geschwindigkeitsgewinn gegenüber sequenziellen Verfahren wie VA-Dateien ist jedoch »nur« etwa Faktor 10 **[Jag05]**, was beim Einsatz in Dateisystemen durch andere Faktoren (→ *3.3.3*) kompensiert wird.

3.3.2 Partial Match Operationen

Da fast alle multidimensionalen Indexstrukturen für die Nachbarschaftssuche entwickelt wurden, werden keine Operationen mit nur wenigen spezifizierten Attributwerten unterstützt, denn mit jedem irrelevanten Attribut wächst bei geometrischer Betrachtung die Dimension des Referenzobjekts. Bei voll spezifizierten Anfragen hat das Referenzobjekt 0 Dimensionen, ist also ein Punkt. Ist nur ein Attribut irrelevant, so entsteht eine eindimensionale Referenzgerade, bei 2 irrelevanten Attributen eine Referenzebene und so weiter. Dies erhöht den Aufwand für die Nachbarschaftssuche enorm, was am Beispiel eines kd-Baums **[Ben75]** erläutert werden soll.

Ein kd-Baum ist ein binärer Baum, dessen Ebenen innerer Knoten zyklisch durch alle Attribute rotieren **[Ben75]**. Als Beispiel dient ein zweidimensionaler Ausschnitt des RGB-Farbraums, bei denen die Farben rot und blau die zwei Dimensionen bilden:

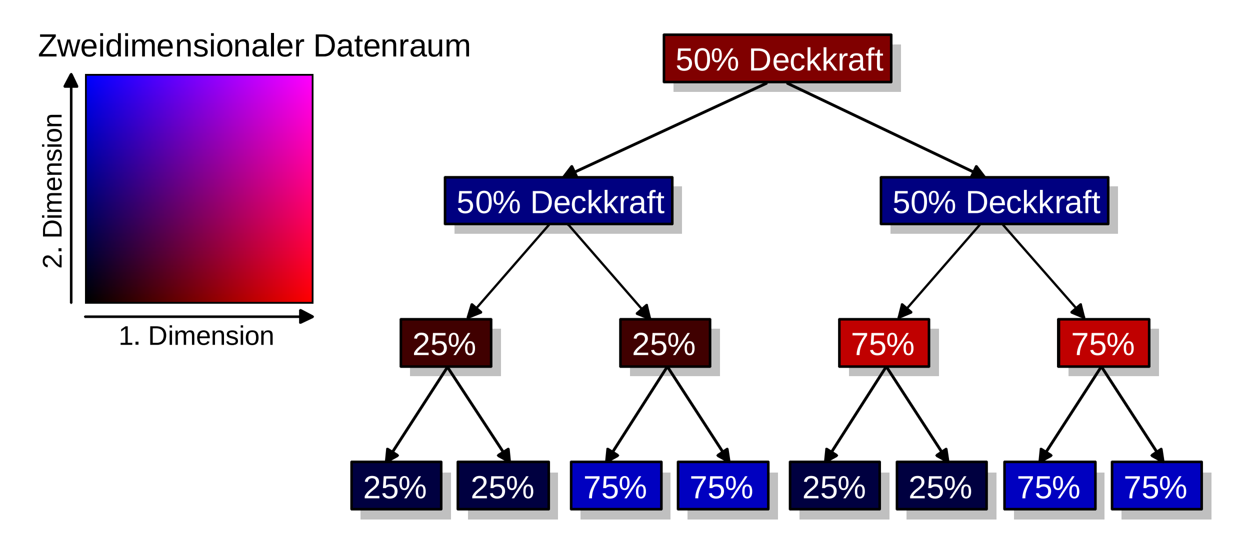


Abb. 3.3-6: Ausschnitt eines kd-Baums über 2 Dimensionen (rot und blau)

In → *Abb. 3.3-6* wird auf der ersten Ebene des kd-Baums die erste Dimension behandelt. Alle Blätter, die über den linken Ast erreichbar sind, haben weniger als 50% rote Farbdeckung, alle über den rechten Ast erreichbaren Farben haben mindestens 50% Rotanteil. Auf der zweiten Ebene wird nun unabhängig von der ersten Dimension über den Farbwert der 2. Dimension (blau) entschieden. Der jeweils links Ast führt zu Farben, die weniger als 50% blauer Farbabdeckung aufweisen, die jeweils rechten Zweige zu Farben mit mindestens 50% Blauanteil – unabhängig vom jeweiligen Rotwert.

Wird nun ein Attribut, etwa der Wert für »blau«, nicht spezifiziert, so müssen von jedem inneren Knoten, der sich auf dieses Attribut bezieht, beide Zweige verfolgt werden, denn es gibt keinen Attributwert aus der Suchanfrage, anhand dessen die Entscheidung für die Traversierung nur eines Teilbaums getroffen werden könnte. Der Aufwand für die Suche wächst somit exponentiell mit der Anzahl irrelevanter Attribute, so dass Partial Match Operationen nicht effizient unterstützt werden.

3.3.3 Ineffizienz von persistent gespeicherten Bäumen

Eine weiteres Problem, das ausnahmslos **alle** Baumstrukturen betrifft, ist das Aufzwingen einer zusätzlichen Zeitkomplexität durch das Speichermedium und das verwendete Dateisystem.

3.3.3.1 Zeitkomplexität durch das Speichermedium

Über einen wichtigen Einfluss auf die Leistungsfähigkeit von Indexen verfügt das Zugriffsmuster der zugrunde liegenden Algorithmen auf das Speichermedium. Als Extrempunkte sind hier ein rein

sequenzieller Zugriff auf hintereinander liegende Sektoren und ein randomisierter Zugriff auf beliebige Sektoren zu unterscheiden. Speichermedien sind ausschließlich auf sequenzielle Operationen hin optimiert, dies gilt bei moderner Hardware sogar abgeschwächt für den RAM-Speicher.

Um dies zu verdeutlichen, wurde die Leistung beider Zugriffsmuster von einem Testprogramm (→ *Kap. B*) auf verschiedenen Datenträgern (mechanischen Festplatten und Flash-Speichern) untersucht. Für beide Zugriffsarten wurden jeweils 10 Durchläufe pro Datenträger ausgeführt, bei jedem Durchlauf wurden 1024 KB, also 2048 Sektoren, eingelesen. Für den sequenziellen Zugriff wurde bei jedem Durchlauf ein zufälliger Startblock ausgewählt (also 10 Mal), beim randomisierten Zugriff für jeden einzelnen Sektor (also 20480 Mal).

Bei allen getesteten Speichermedien war der sequenzielle Zugriff deutlich schneller, maximal bis etwa Faktor 300 (→ *Kap. B.2*) . Dies gilt sogar für den Flash-Speicher, da hier beim randomisierten Lesen für jeden Sektor neue Adressen an den Speicher-Controller übertragen werden müssen.

Dieser Effekt impliziert einen Geschwindigkeitsnachteil für Baumstrukturen, da beispielsweise beim Verfolgen eines Pfades durch einen B-Baum für jeden Knoten ein neuer Sektor vom Datenträger eingelesen werden muss, und dieser in der Regel nicht unmittelbar hinter dem Vorgänger liegt. In der Praxis werden dabei allerdings nie Einbußen um einen so hohen Faktor erlitten, da sich die meisten Sektoren der Indexstruktur zumindest im selben Gebiet der Plattenoberfläche befinden und nicht völlig zufällig verteilt sind. Aber selbst dann, bzw. sogar bei Flash-Speicher ohne mechanische Teile, bietet das sequenzielle Lesen einen Geschwindigkeitsvorteil.

3.3.3.2 Zeitkomplexität durch das Dateisystem

Zusätzlich zu den unvermeidlichen Leistungseinbußen, den Speichermedien bei wahlfreiem Zugriff im Gegensatz zu sequenziellem Zugriff haben, verringert auch das verwendete physikalische Dateisystem die Zugriffsgeschwindigkeit je nach Typ drastisch.

Wird zum Beispiel ein B-Baum innerhalb einer Datei gespeichert, hängt die Traversierungsgeschwindigkeit zusätzlich davon ab, mit welchem Zeitaufwand der Dateizeiger auf eine neue Stelle der Indexdatei bewegt werden muss. Ältere Dateisysteme wie *FAT*, die Datenblöcke linear verketten, verlangsamen das Traversieren eines Baums um $O(n)$ auf $O(n \log n)$, während bei ext2 ein Dateizeiger durch eine mehrstufige Blockadressierung **[Kol07b]** in konstanter Zeit positioniert werden kann.

Diese Abhängigkeit von der Leistung des physikalischen Dateisystems stellt schon seit längerer Zeit ein Problem dar. Datenbanksysteme operieren daher oft auf unformatierten Speichermedien ohne Dateisystem, wodurch ein wahlfreier Zugriff auf beliebige Sektoren nur vom verwendeten Datenträger abhängt (→ *Kap. 3.3.3.1*).

3.3.4 Partiell belegter Datenraum

Eine besonders nützliche Eigenschaft des Datenraums, den Dateiattribute bilden, wird von multidimensionalen Indexstrukturen bisher nicht explizit unterstützt: sie gehen in der Regel davon aus, dass sich ein Objekt an jedem Punkt im Datenraum Ω befinden kann. Speziell bei der Metadaten-Indexierung ist dies jedoch nicht gegeben, da Dateiformate nur bestimmte Metadaten enthalten: so hat eine MP3-Datei beispielsweise keine Breite oder Höhe **[Nil05]**, während der Exif-Standard keinen Albumnamen und kein Genre kennt **[EXI07]**. Es existieren also in der Praxis keine Dateien, die alle dem Dateisystem bekannten Metadaten-Attribute mit Werten belegen – in den entsprechenden Regionen eines Datenraums können also keine Dateien enthalten sein (»sparsity«). Der Objektraum ω hat somit **nicht** die Form des Hyperquaders Ω **[Kol08b] [Kol08c]**:

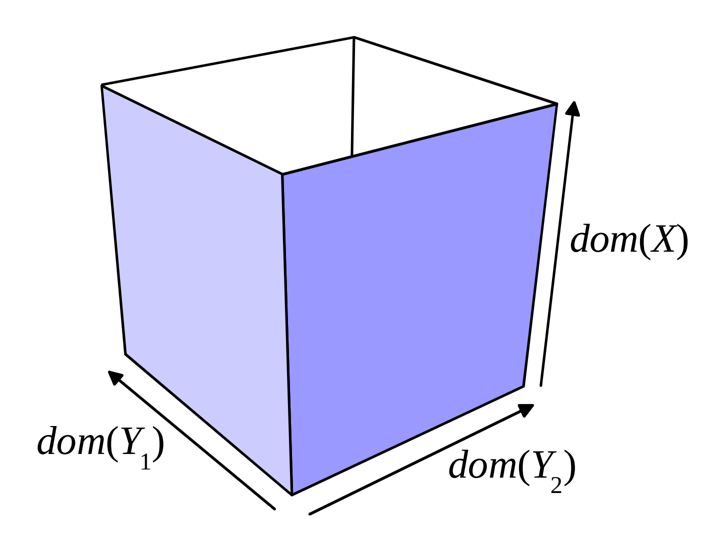


Abb. 3.3-7: Objektraum ω für zwei Dateiformate mit je zwei Attributen (blau)

In → *Abb. 3.3-7* wird der Datenraum Ω Hyperquader dargestellt. Dateiformat 1 liefere die Attribute X und Y_1, Dateiformat 2 die beiden Attribute X und Y_2. Ausgehend von diskreten Wertebereichen für alle Attribute lässt sich der Hyperquader als $\Omega = dom(X) \times dom(Y_1) \times dom(Y_2)$ beschreiben. Der Objektraum ω ist in diesen Hyperquader eingebettet und besteht nur aus den zwei blau eingefärbten Flächen **[Kol08b] [Kol08c]**:

$$\omega = \left(dom(X) \times dom(Y_1)\right) \cup \left(dom(X) \times dom(Y_2)\right) \tag{3.3}$$

Der Objektraum kann auch allgemein für mehr als zwei Dateiformate und beliebige Attributmengen für jedes Format definiert werden. Es seien k Dateitypen definiert, die zusätzlich zu den n Attributen X_1 bis X_n, die in allen Formaten vorkommen, jeweils m weitere Attribute haben, die von $Y_{j,1}$ bis Y_{j,m_j} durchnummeriert werden. Dann gilt **[Kol08b] [Kol08c]**:

$$\omega = \bigcup_{j=1}^{k} dom(X_1) \times \ldots \times dom(X_n) \times dom(Y_{j,1}) \times \ldots \times dom(Y_{j,m_j}) \tag{3.4}$$

3.3.5 Performanz

In den vorhergehenden Abschnitten wurde dargelegt, welche theoretischen Schwierigkeiten beim Einsatz klassischer multidimensionaler Indexe zum Speichern von hochdimensionalen Dateiattributen zu erwarten sind:

1. Degenerieren von Datenstrukturen, die den Daten- oder Objektraum partitionieren, bei vielen Dimensionen (»Fluch der Dimensionen« ,→ *Kap. 3.3.1*)
2. Keine Unterstützung von Partial Match Operationen, die gerade im Zusammenhang mit Dateiattributen besonders häufig vorkommen (→ *Kap. 3.3.2*)
3. Ineffizienz von persistent gespeicherten Bäumen (→ *Kap. 3.3.3*)
4. Keine Ausnutzung des partiell belegten Datenraums (→ *Kap. 3.3.4*)

Um zu zeigen, dass diese theoretischen Überlegungen auch Auswirkungen in der Praxis haben, wurde die Leistung derartiger Indexstrukturen anhand von zwei relationalen DBMS getestet: dem Microsoft SQL-Server 2005 (→ *Kap. 3.3.5.1*) und dem quelloffenen DBMS »PostgreSQL« (→ *Kap. 3.3.5.2*). Das Testsystem war mit einer AMD Athlon 64 X2 3800 dual core CPU (2000 MHz), 2 GB DDR2 dual channel RAM (200 MHz Bustakt) und einer 320 GB SATA-Festplatte ausgestattet, die als *NTFS*-Dateisystem formatiert wurde. Auf diesem System wurden die Bearbeitungszeiten von fünf Suchanfragen ermittelt, die auf einem genau definierten Datenbestand mit 62 Dimensionen ausgeführt wurden:

1. Der erste Testfall sucht alle Dateien, bei denen ein Bit für »Neue Datei« gesetzt ist. Auf den Testdaten findet diese Anfrage jedoch keine Dateien.
2. Der zweite Fall soll alle Bilddateien finden. Das Finden von Dateien eines bestimmten Typs ist eine grundlegende Funktion, auf der alle Suchen nach typabhängigen Metadaten basieren.
3. Der dritte Testfall ist analog zum vorigen Fall konstruiert und soll alle Audio-Dateien finden.
4. Die vierte Suchanfrage erweitert den zweiten Testfall: es sollen nicht mehr alle Bilddateien gefunden werden, sondern nur noch diejenigen, deren Breite ≥ 1024 Pixel ist. Die Bildauflösung ist bei allen Bilddateien am Dateianfang zu finden **[EXI07]**.
5. Der fünfte Suchfilter erweitert den dritten Testfall: es wird nun nur noch nach denjenigen Audio-Dateien gesucht, die in ihren Metadaten als Künstler »Anastacia« eingetragen haben. Der ID3-Tag **[Nil05]** befindet sich immer am Dateiende.

3.3.5.1 Microsoft SQL Server 2005

Der Microsoft SQL-Server 2005 ist als Standard-Lösung von besonderem Interesse, da das Prestige-Produkt Microsoft WinFS (→ *Kap. 2.9*) dieses Datenbanksystem zum Speichern von Dateien einsetzt. Um die Performanz des Microsoft SQL-Servers zu testen, wurde nach der Installation über das »Management Studio« eine neue Datenbank mit dem Namen **Metadata** angelegt:

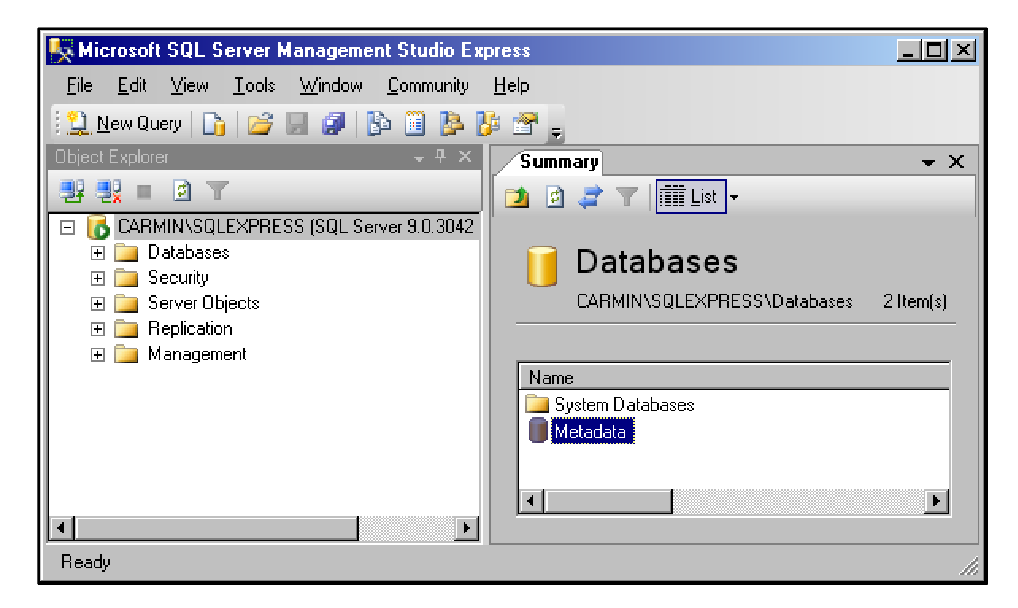


Abb. 3.3-8: Angelegte Datenbank ***Metadata***

Innerhalb einer Datenbank können mehrere Tabellen, Indexe, Logs usw. abgespeichert werden. Der weitere Zugriff, insbesondere das Anlegen einer geeigneten Tabelle und das Erstellen von Indexen, wurde durch das Testprogramm in → *Kap. C* durchgeführt. Besonders problematisch ist die geringe Konfigurierbarkeit der Indexe. Ein Index kann höchstens 16 Attribute umfassen, so dass keine wirklich hochdimensionalen Indexe (→ *Kap. 3.3*) angelegt werden können:

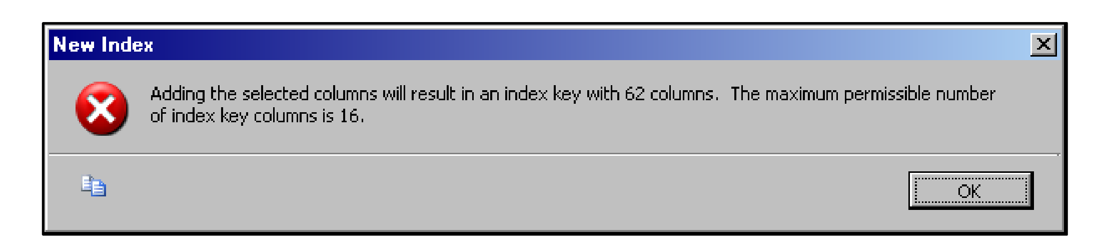


Abb. 3.3-9: Begrenzte Indexierung (max. 16 Attribute)

Statt dessen wurden mehrere Indexe mit den jeweils maximal 16 zulässigen Dimensionen erstellt. Ein Index wurde als »clustered index« angelegt (Primärindex, der die physikalische Sortierung beeinflusst), alle anderen Indexe waren »unclustered«. In dieser Konfiguration wurde die Ausführungszeit für die fünf Suchanfragen durch ein Testprogramm ermittelt.

Als Hauptaussage ist dabei zu treffen, dass eine Suche mit Index länger dauert als ohne Indexierung (→ *Abb. 3.3-10*) – also genau anders als eigentlich beabsichtigt. Dadurch ist der Microsoft SQL-Server ungeeignet für die Indexierung von Metadaten, was die bereits festgestellten Mängel von Microsoft WinFS (→ *Kap. 2.9.2*) erklärt. Zusätzlich führt die Indexierung noch zu einem Geschwindigkeitsverlust beim Hinzufügen neuer Tupel (→ *Abb. 3.3-11*).

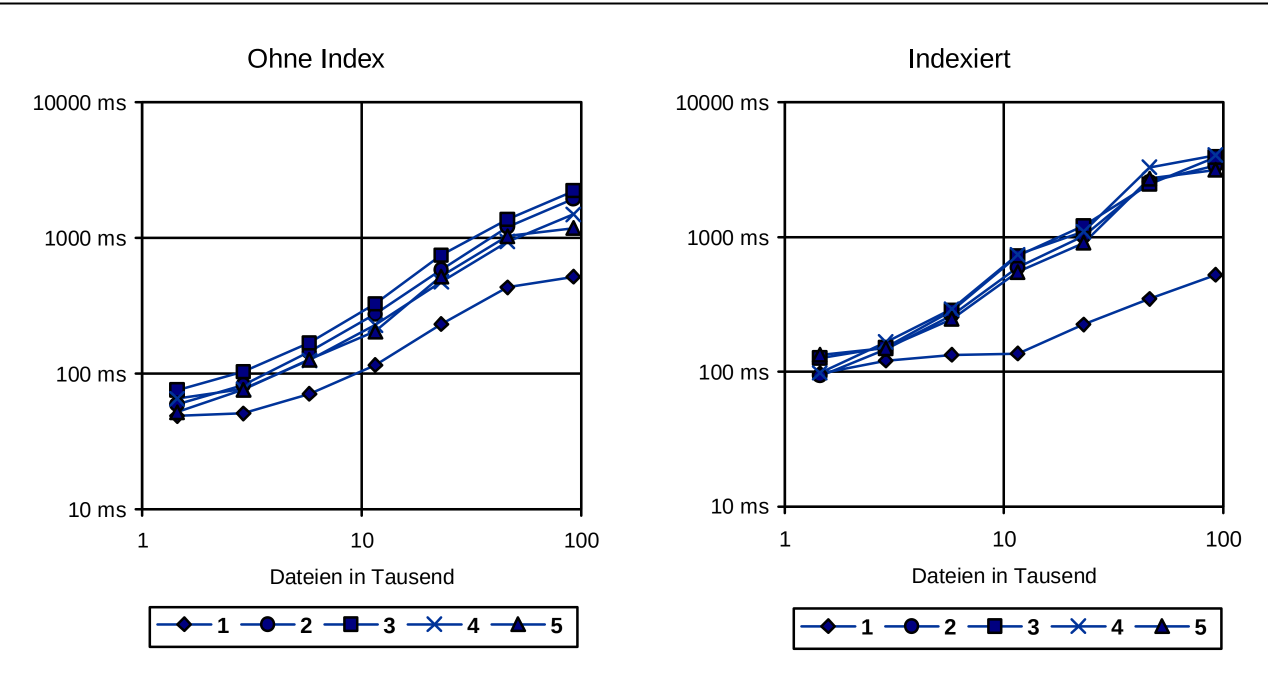


Abb. 3.3-10: Leistungsfähigkeit des Microsoft SQL Server **[Kol08c]**

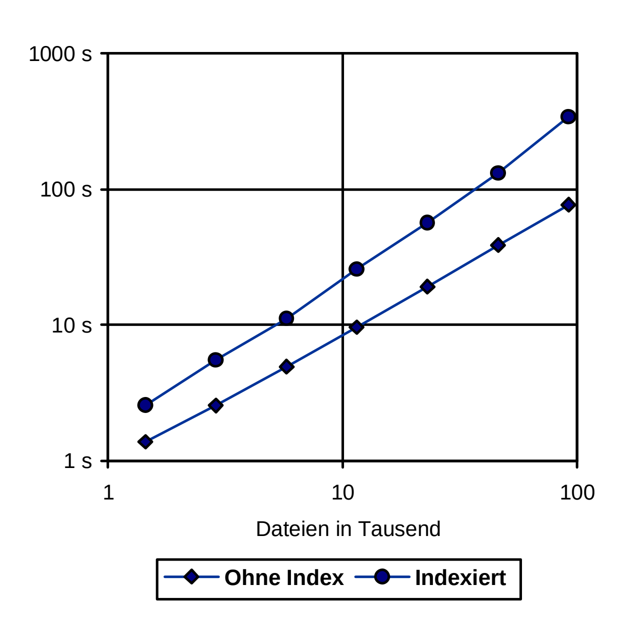


Abb. 3.3-11: Ausführungszeiten für XML bulk loading

3.3.5.2 PostgreSQL

Ähnliche Effekte wie der im vorigen Abschnitt untersuchte Microsoft SQL-Server 2005 zeigt auch das quelloffene DBMS »PostgreSQL«. Analog zu → *Kap. C* wurde ein Testprogramm (→ *Kap. D*) eingesetzt, um die Leistung von PostgreSQL beim Indexieren von Metadaten zu messen. PostgreSQL-Datenbanken können unterschiedliche Indexstrukturen einsetzen, allerdings sind nur B-Bäume für hochdimensinale Datenräume mit heterogenen Datentypen (z.B. String und int) geeignet. Eine Datenbank mit den Attributen von 92288 Dateien und einem Index für jedes Attribut belegte 101 MB:

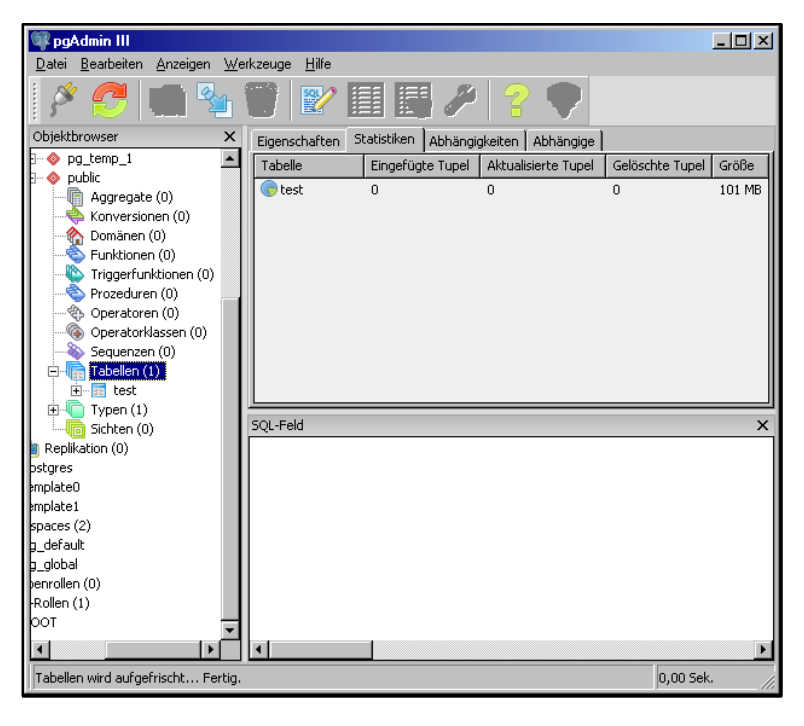


Abb. 3.3-12: Größe einer Datenbank für 92.288 Dateien

Analog zu → *Kap. 3.3.5.1* wurden die Laufzeiten für dieselben fünf Suchanfragen ermittelt. Indexe in Form von B-Bäumen für jedes Attribut boten dabei keinen oder keinen nennenswerten Vorteil:

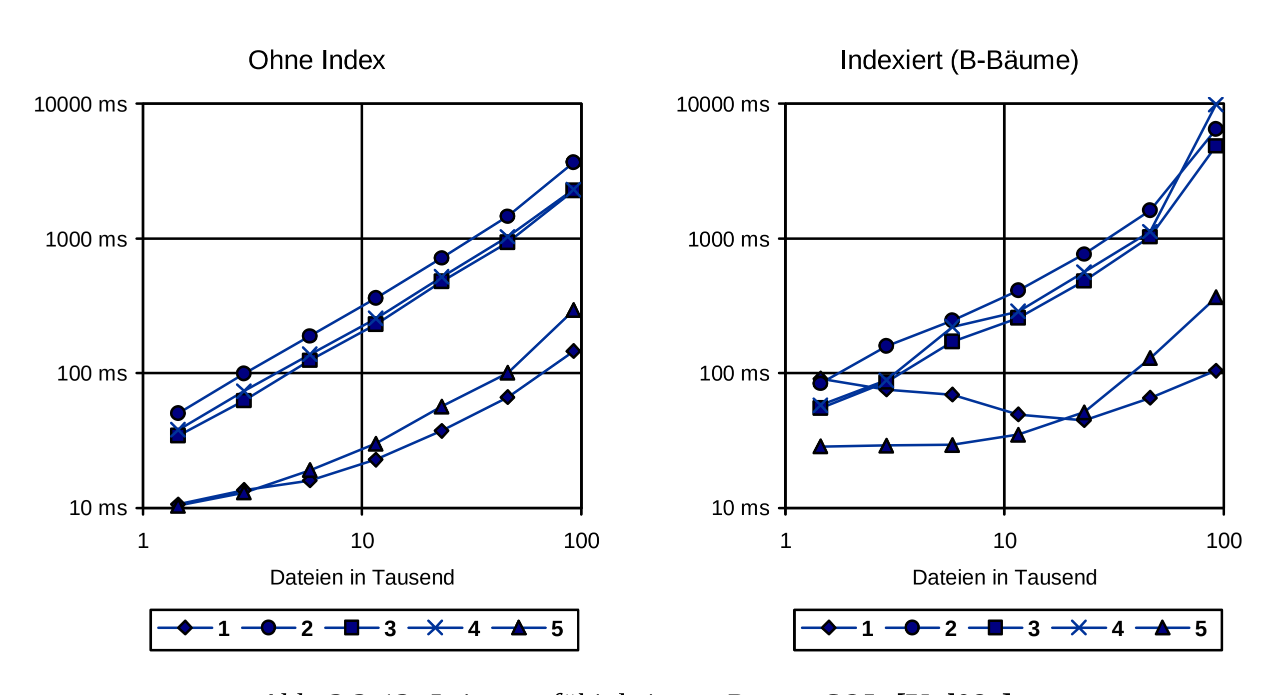


Abb. 3.3-13: Leistungsfähigkeit von PostgreSQL **[Kol08c]**

Der marginale Zeitvorteil wird jedoch durch eine deutlich geringere Performanz beim Hinzufügen von neuen Tupeln zum Index erkauft. Das ist insbesondere beim Einsatz in Dateisystemen ein Problem, so dass die theoretischen Überlegungen und Aussagen (→ *Kap. 3.2.1*) der Entwickler des BeOS File System (→ *Kap. 2.7*) durch diese Messungen noch einmal bestätigt werden:

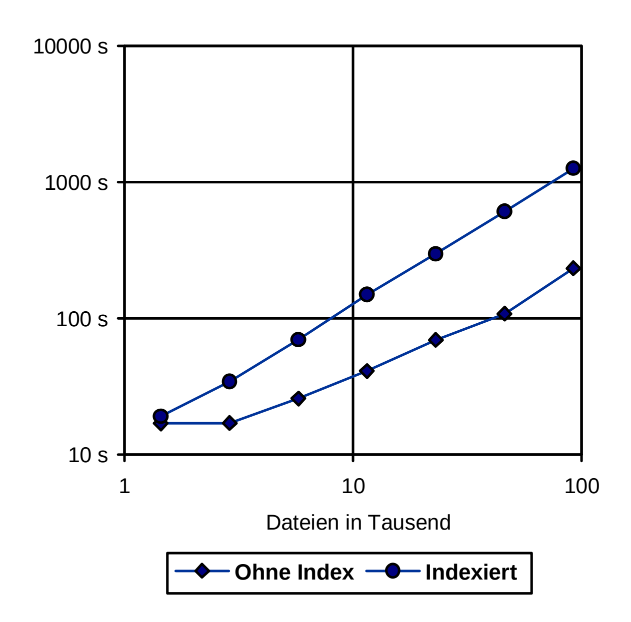


Abb. 3.3-14: Ausführungszeiten für XML bulk loading

3.4 Fazit

In diesem Kapitel wurden mehrere Ansätze zur Indexierung vorgestellt. Leider haben alle betrachteten Verfahren gravierende Nachteile, die einem Einsatz zur Indexierung von Metadaten innerhalb eines Dateisystems entgegenstehen.

Mit der Lucene-Bibliothek (→ *Kap. 3.1*) steht Anwendungsprogrammen eine generische Bibliothek zur Indexierung mit invertierten Listen zur Verfügung, die unter anderem von einigen Desktop-Suchmaschinen (→ *Kap. 2.10*) eingesetzt wird **[Luc07]**. Da an einem Dateisystem häufig Änderungen vorgenommen werden (z.B. bei jedem Speichern einer Datei), muss ein Index nicht nur schnelle Suchergebnisse liefern, sondern sich auch mit geringem Aufwand aktualisieren lassen. Lucene leistet dies nicht: aufgrund der invertierten Listen sind Aktualisierungen sogar sehr aufwändig, da zusätzlich zu den eigentlichen Informationen in den Felddaten (Dateiendung **.fdt**) für jeden Attributwert Einträge im Feldindex (Dateiendung **.fdx**) zu modifizieren sind (→ *Abb. 3.1-3*).

Die bei Suchanfragen über Dateien häufig auftretenden Partial Match Operationen (→ *Kap. 3.3.2*) werden effizient von einem Wald aus B- oder B^+-Bäumen unterstützt, wie sie von BeFS (→ *Kap. 2.7*) eingesetzt werden. Leider tritt hier bei Aktualisierungen dasselbe Problem wie oben beschrieben auf: bei jeder Änderung an einer Datei muss ein Baum pro Attribut modifiziert werden. Aus diesem Grund indexiert das BeFS nicht alle Attribute (→ *Kap. 3.2.1*). Die Performanz verschlechtert sich weiter, da durch die nicht-sequenziellen Zugriffe auf Baumstrukturen das eingesetzte Speichermedium zu großen Geschwindigkeitsverlusten führen kann (→ *Kap. 3.3.3*).

Partial Match Operationen werden auch von den meisten multidimensionalen Indexstrukturen (→ *Kap. 3.3*), die alle Attribute in einer Datenstruktur zusammenfassen, nicht effizient unterstützt,

da diese vor allem für Nachbarschaftssuchen entwickelt wurden. Somit ist insbesondere der Einsatz von kd- und kdB-Bäumen (→ *Kap. 3.3.2*) sowie von Gridfiles **[Saa05]** und iDistance **[Jag05]** ausgeschlossen (rechte Spalte in → *Abb. 3.3-2*). Multidimensionale Indexe, die den Datenraum Ω oder den Objektraum ω partitionieren, degenerieren darüber hinaus aufgrund des »Fluchs der Dimensionen« beim Einsatz auf hochdimensionalen Daten **[Web98]**. Als Konsequenz besteht das Problem eines nicht-sequenziellen Zugriffs auch bei den meisten multidimensionale Indexstrukturen, wodurch sogar der Geschwindigkeitsgewinn von iDistance (→ *Kap. 3.3.1*) auf nicht gleichverteilten Daten wieder vernichtet wird.

Als einziges Verfahren bleibt für das Indexieren der Attribute somit nur das sequenzielle Lesen aller Attributwerte aus einer geeigneten Datei. Statt eine bereits in der Theorie verlorene Schlacht zu schlagen, wird in → *Kap. 5* mit dem »Master/Slave-Index« eine Indexstuktur auf Basis einer sequenziellen Suche beschrieben, die außerdem den partiell belegten Datenraum (→ *Kap. 3.3.4*) ausnutzt.

4 Integration

Im Rahmen dieses Kapitels wird gezeigt, dass sich die Datenmodelle von BLOB-relationalen Datenbanken (→ *Kap. 4.2.1*) und semantischen Dateisystemen (→ *Kap. 4.3.1*) auf das »Library«-Modell (→ *Kap. 4.1*) abbilden lassen, und sich umgekehrt das Modell einer Library auf beide Datenmodelle abbilden lässt. Es wird gezeigt, wie Abbildungsvorschriften konstruiert werden können, durch die keine Informationen verloren gehen. Dadurch können BLOB-relationale Datenbanken und semantische Dateisysteme als gleich mächtig und in diesem Sinne äquivalent angesehen werden.

Mittels der Integration beider Datenmodelle wird ein Problem gelöst, auf das Shoens et al. bereits 1993 in **[Sho93]** hingewiesen haben: da die Möglichkeiten traditioneller Dateisysteme begrenzt sind, implementieren viele Applikationen proprietäre Datenbanken, um ihre Nutzdaten dort abzuspeichern. Diese Datenbanksysteme, und damit die zugehörigen Applikationen, nehmen die Daten jedoch »in Besitz«, d.h. sie sind für andere Programme über das Dateisystem nicht mehr zugänglich. Ein Beispiel für diesen Sachverhalt sind EMail-Programme, die neben dem eigentlichen Textkörper auch Attribute wie Absender, Datum und Thema der EMail verwalten müssen:

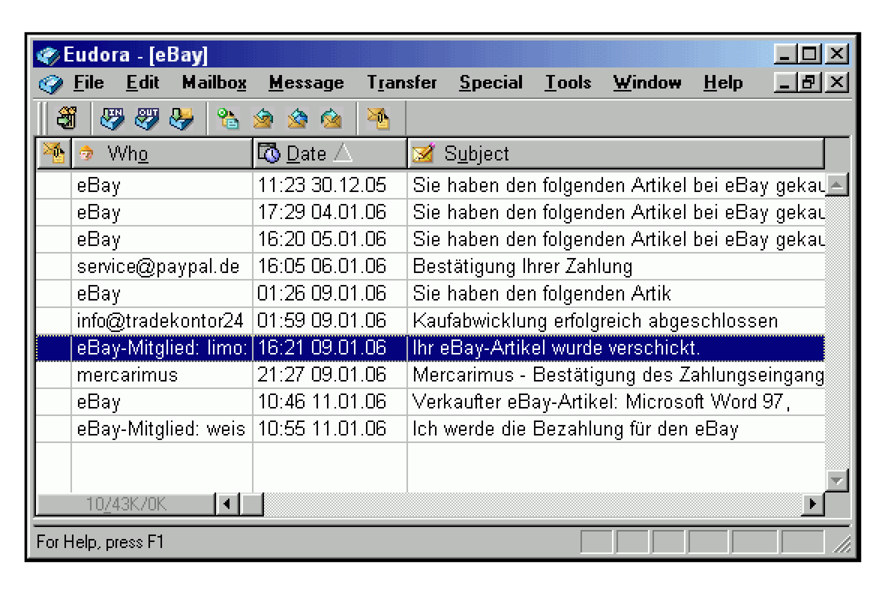


Abb. 4-1: Qualcomm Eudora mit geöffnetem Postfach

Wird nun mit herkömmlichen Mitteln nach Dateien mit einem bestimmten Schlagwort gesucht, so würde lediglich der Name der Datenbank-Datei im Suchergebnis erscheinen (in → *Abb. 4-1* also die Datei EBAY.MBX), nicht jedoch ein individuelles Datenobjekt wie z.B. eine bestimmte EMail.

Die Begriffe »Datenbank« und »Dateisystem« werden hier angelehnt an die ANSI-SPARC-Architektur **[Saa05]** im Sinne von → *Abb. 4-2* eingesetzt: eine Datenbank bzw. ein Dateisystem sind logische Strukturen, die auf einer physikalischen Ebene umgesetzt werden. Ein DBMS kann mehrere Datenbanken verwalten, und bietet Operationen auf diesen an. Für Dateisysteme stellen Betriebssysteme und die Firmware von Multimedia-Geräten entsprechende Funktionen bereit:

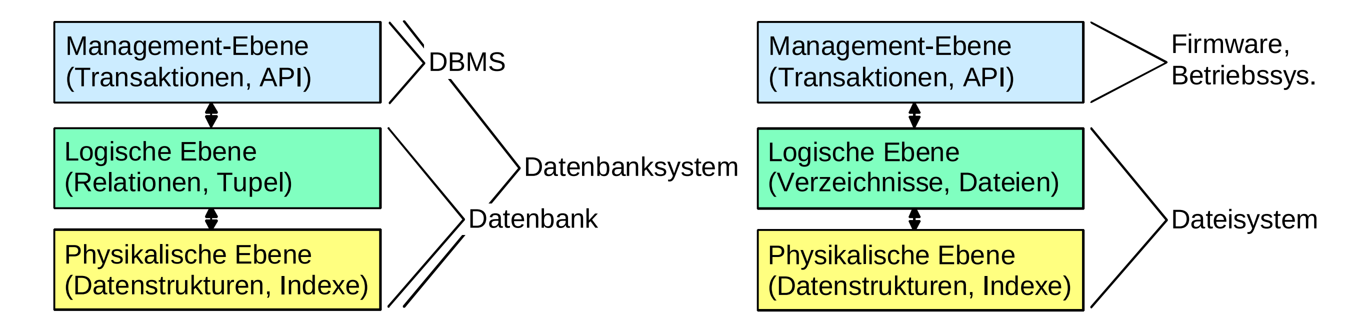


Abb. 4-2: Physikalische, logische und Management-Ebene verschiedener Speichersysteme

Leider wird der Begriff »Datenbank« sowohl umgangssprachlich als auch in der Literatur oft stellvertretend für »Datenbanksystem« und DBMS verwendet, was jedoch im weiteren Verlauf zu Fehlschlüssen führen würde.

4.1 Libraries

In diesem Abschnitt wird das Datenmodell einer Library eingeführt. Basierend auf dem relationalen Datenmodell **[Cod70]** wird zunächst ein Attribut A definiert. Jedem Attribut A sind eine Bezeichnung, welche die Bedeutung des Attributs beschreibt, und mit $dom(A)$ die Gesamtheit aller Werte, die A enthalten kann, zugeordnet. Üblicherweise entspricht $dom(A)$ einem atomaren Datentyp, aber auch insbesondere beliebig lange Bytefolgen (Dateikörper bzw. *BLOBs*) sind in Libraries möglich. Sie werden hier ebenfalls als atomar angesehen. Darüber hinaus seien Nullmarken für alle Wertebereiche zulässig **[Zan82]**.

Ein Objekt-Schema R wird als Menge von Attributen definiert. Jedes Schema enthält ein ausgezeichnetes Attribut F mit Primärschlüsselfunktion (4.5) sowie ein Attribut T, das den Typ aller Datenobjekte eindeutig codiert (4.6). Zusätzlich enthalten sind d weitere Attribute $A_1 \dots A_d$ mit $d > 0$:

$$R = \{F, T, A_1, \dots, A_d\},\ d > 0 \tag{4.1}$$

Ein Objekt-Schema R erzeugt einen Datenraum Ω, der aus dem kartesischen Produkt der Wertebereiche $dom(X, R)$ aller Attribute X von R besteht:

$$\Omega(R) = dom(R.F) \times dom(R.T) \times dom(R.A_1) \times \dots \times dom(R.A_d) \tag{4.2}$$

Ein Datenobjekt o wird als geordnetes Tupel von Attributwerten definiert, das einem vorher definierten Schema R genügt, und daher in $\Omega(R)$ enthalten ist:

$$\begin{aligned} &o \in \Omega(R) \\ &o = \left\langle f, t, a_1, \dots, a_d \;\middle|\; f \in dom(R.F) \wedge t \in dom(R.T) \wedge a_i \in dom(R.A_i) \wedge f, t \neq Nullmarke \right\rangle \end{aligned} \tag{4.3}$$

Datenobjekte sind oft hochdimensional, da die Anzahl der Attribute eines Objekt-Schemas in der Praxis sehr groß werden kann: mehr als 50 Attribute sind durch gängige Standards für Metadaten leicht erreichbar **[EXI07] [Nil05]**. Indexstrukturen (→ *Kap. 3*, → *Kap. 6*) werden bereits ab 10 Attributen als hochdimensional bezeichnet **[Web98]**.

Eine Instanz r eines Schemas R ist eine Menge (jedoch keine Multi-Menge) von Datenobjekten und damit eine Teilmenge von $\Omega(R)$:

$$r(R) = \{o_1,\ldots,o_n\} \subseteq \Omega(R) \tag{4.4}$$

Innerhalb einer Instanz $r(R)$ muss die Primärschlüsseleigenschaft von F hergestellt werden, das heißt zwei beliebige nicht-identische Datenobjekte o_1 und o_2 aus $r(R)$ müssen sich zumindest in ihren Attributwerten f unterscheiden (π bezeichne die Projektion der Relationenalgebra):

$$\forall o_1, o_2 : o_1 \in r(R) \wedge o_2 \in r(R) \wedge o_1 \neq o_2 \Rightarrow \pi_F(o_1) \neq \pi_F(o_2) \tag{4.5}$$

In (4.1) wurde gefordert, dass T den Typ aller Datenobjekte eindeutig codiert. Da der Typ nach (4.2) und (4.3) durch das Objekt-Schema R vorgegeben wird, müssen alle Datenobjekte aus $r(R)$ denselben Attributwert für T besitzen, der zusätzlich als Typbezeichner für R angesehen werden kann:

$$\forall o_1, o_2 : o_1 \in r(R) \wedge o_2 \in r(R) \Rightarrow \pi_T(o_1) = \pi_T(o_2) \tag{4.6}$$

Ein Library-Schema L wird definiert als Menge von k Objekt-Schemata $R_1 \ldots R_k$, deren jeweilige Attribute F und T denselben Wertebereich $dom(F)$ bzw. $dom(T)$ umfassen:

$$L = \left\{ R_1,\ldots,R_k \;\middle|\; \begin{array}{c} dom(R_1.F) = \ldots = dom(R_k.F) \\ dom(R_1.T) = \ldots = dom(R_k.T) \end{array} \right\} \tag{4.7}$$

Der Datenraum $\Omega(L)$ eines Library-Schemas L wird durch das kartesische Produkt aller Attribute gebildet. Da die Attribute F und T, aber auch andere Attribute, in mehreren Objekt-Schemata vorkommen können (→ *Kap. 3.3.4*), wird mit $glob(L)$ zunächst die globale Menge aller verfügbaren Attribute als Vereinigung der Objekt-Schemata gebildet. Jedes dieser Attribute bildet eine Dimension des Datenraums $\Omega(L)$:

$$\begin{aligned} glob(L) &= \bigcup_{j=1}^{k} R_j \\ \Omega(L) &= dom(g_1) \times \ldots \times dom(g_{|glob(L)|}) \end{aligned} \tag{4.8}$$

Mit $common(L)$ sei die Menge aller Attribute bezeichnet, die in allen R_j aus L enthalten sind. Diese Menge ist somit die Schnittmenge aller R_j, und enthält aufgrund (4.1) insbesondere F und T. Datenobjekte können in $\Omega(L)$ nur in ausgewählten Regionen existieren (partiell belegter Datenraum,

→ *Kap. 3.3.4*). Als X_i seien die n Attribute bezeichnet, die in allen k Objekt-Schemata vertreten sind, $Y_{j,i}$ seien die übrigen m_j Attribute aus R_j. Für den Objektraum $\omega(L)$ gilt dann (→ *Kap. 3.3.4*):

$$common(L) = \bigcap_{j=1}^{k} R_j$$

$$\omega(L) = \bigcup_{j=1}^{k} dom(X_1) \times \ldots \times dom(X_n) \times dom(Y_{j,1}) \times \ldots \times dom(Y_{j,m_j}), \quad \begin{array}{l} X_i \in common(L) \wedge \\ Y_{j,i} \in R_j - common(L) \end{array} \tag{4.9}$$

Die Attribute F und T sind damit immer Elemente von $common(L)$. Eine Instanz l eines Library-Schemas L ist nun eine Menge von Datenobjekten und eine Teilmenge von $\omega(L)$:

$$l(L) = \{o_1, \ldots, o_{|l(L)|}\} \subseteq \omega(L) \tag{4.10}$$

Jedes Datenobjekt o eines Objekt-Schemas $R \in L$ kann zum Element o' des globalen Datenraums $\Omega(L)$ werden, indem die Werte der im zugehörigen Objekt-Schema R nicht vorhandenen Attribute durch Nullmarken besetzt werden (Verwendung von ≥ nach **[Zan82]**):

$$o \in \Omega(R) \wedge R \in L \Rightarrow \exists o' \in \Omega(L), o' \geq o \tag{4.11}$$

Innerhalb einer Library $l(L)$ müssen sich zwei beliebige Datenobjekte o_1 und o_2 aus $l(L)$, die nicht identisch sind, zumindest in ihren Attributwerten $\langle f, t \rangle$ unterscheiden, wodurch diese Attribute einen global gültigen Superschlüssel bilden:

$$\forall o_1, o_2 : o_1 \in l(L) \wedge o_2 \in l(L) \wedge o_1 \neq o_2 \Rightarrow \pi_{F,T}(o_1) \neq \pi_{F,T}(o_2) \tag{4.12}$$

Auf dem Library-Modell können alle Operationen der Relationen-Algebra durchgeführt werden, sofern keine der oben dargestellten Vorschriften, insbesondere (4.5), (4.6) und (4.12), verletzt werden.

4.2 Relationale Datenbanken

In diesem Abschnitt wird gezeigt, dass das relationale Datenmodell ohne Verlust von Informationen auf das Datenmodell einer Library abgebildet werden kann. Da sowohl die Schemata relationaler Datenbanken als auch Library-Schemata eine Menge von Relations- bzw. Objekt-Schemata sind, genügt es zunächst, eine Abbildungsvorschrift zu konstruieren, die Relations-Schemata durch ein Schema-Mapping **[Les06]** in Objekt-Schemata einer Library überführt.

Ein Relations-Schema R' ist genau wie ein Objekt-Schema R eine Menge von Attributen $A_1 \ldots A_{d'}$. Neben diesen allgemeinen Attributen fordern Objekt-Schemata jedoch ein Schlüsselattribut F und ein Typ-Attribut T. Diese müssen von einer Abbildungsvorschrift hinzugefügt werden, so dass gilt:

$$R = R' \cup \{F, T\} = \{F, T, R'.A_1, \ldots R'.A_{d'}\} \tag{4.13}$$

Alle Attribute aus R' werden unverändert übernommen, so dass bei der Abbildung keine Informationen verloren gehen. Für die Attribute F und T muss die Bedingung (4.7) eingehalten werden. Bei der Abbildung $r(R') \rightarrow r(R)$ müssen die Attributwerte f und t unter Beachtung der Vorschriften (4.3), (4.5), (4.6) und (4.12) vergeben werden.

4.2.1 BLOB-relationale Datenbanken

Auch in umgekehrter Richtung $R \rightarrow R'$ kann eine Abbildung konstruiert werden, so dass Objekt-Schemata R in Relations-Schemata R' umgewandelt werden können. Da Relations-Schemata beliebige Attribute enthalten dürfen, können sowohl alle Attribute A als auch das Schlüsselattribut F und das Typ-Attribut T in ein Relations-Schema aufgenommen werden.

In diesem Zusammenhang ist es problematisch, dass für die Attribute von Objekt-Schemata BLOBs als Wertebereich explizit zugelassen sind. Eine Library **behandelt** diese zwar als atomar, das klassische relationale Datenmodell hingegen gestattet keine BLOBs, da diese in Wirklichkeit eben nicht atomar sind.

In der Praxis entstand jedoch schon früh der Wunsch, auch Binärdaten in einer Relation abzuspeichern. Als Beispiel können Fotos in einer Relation mit Personen dienen, oder eingescannte Titelblätter in einer Relation für Zeitschriften. Moderne Datenbanksysteme gestatten daher auch das Speichern von BLOBs, und sehen diese ebenfalls als atomar an.

Aus diesem Grund kann eine Trennung zwischen relationalen Datenbanken ohne und mit Unterstützung für BLOBs gezogen werden. Letztere werden hier »BLOB-relational« genannt. Damit lassen sich alle relationalen Datenbanken auf eine Library abbilden, Libraries hingegen können allgemein nur auf BLOB-relationale Datenbanken abgebildet werden:

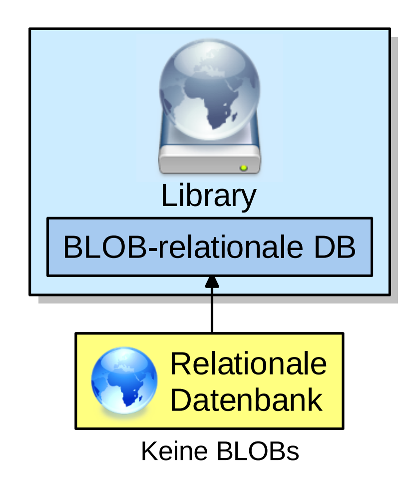


Abb. 4.2-1: Relationale Datenbank, BLOB-relationale Datenbank und Library

4.3 Dateisysteme

In diesem Abschnitt wird gezeigt, dass das Datenmodell von Dateisystemen ohne Verlust von Informationen auf das Datenmodell einer Library abgebildet werden kann. Ein Datei-Schema R' ist eine Menge, die abhängig vom Dateisystem aus einem oder mehreren Dateikörpern $K_1 \ldots K_{d''}$ besteht, und zusätzlich weitere Attribute $A_1 \ldots A_{d'}$ enthält:

$$R' = \{A_1, \ldots, A_{d'}, K_1, \ldots, K_{d''}\},\ d' > 0 \wedge d'' > 0 \qquad (4.14)$$

Übliche Attribute sind beispielsweise der Dateiname, der Zeitpunkt der letzten Änderung und die Größe des Dateikörpers. Die meisten Dateisysteme können für jede Datei nur einen einzigen Dateikörper abspeichern, moderne Dateisysteme wie z.B. NTFS gestatten jedoch auch das Speichern beliebig vieler sog. »Forks« für jede Datei **[Kol07b]**.

Das Schema traditioneller Dateisysteme enthält nur ein einziges Datei-Schema, dem alle Dateien unabhängig von ihrem Format entsprechen müssen, so dass gilt:

$$DS = \{R'\} \qquad (4.15)$$

Die folgende Abbildung zeigt ein Beispiel für eine Datei, die einen Dateikörper enthält, und dem Datei-Schema R' = {*Dateiname*, *Geändert am*, *Größe*, *Dateikörper*} genügt:

Dateiname	Geändert am	Größe	Dateikörper
BILD.JPG	08.12.2005	89237 Byte	

Abb. 4.3-1: Beispieldatei

4.3.1 Abbildungsvorschrift

Um ein Dateisystem auf eine Library abzubilden, genügt es analog zu → *Kap. 4.2* zu zeigen, dass sich ein Datei-Schema R' auf ein Objekt-Schema R abbilden lässt. Ein Datei-Schema R' ist genau wie ein Objekt-Schema R eine Menge von Attributen, so dass die allgemeinen Attribute A_1 bis $A_{d'}$ sowie alle d'' Dateikörper problemlos abgebildet werden können. Darüber hinaus müssen das Schlüsselattribut F und ein Typ-Attribut T von einer geeigneten Abbildungsvorschrift unter Beachtung der Vorschriften (4.3), (4.5), (4.6), (4.7) und (4.12) hinzugefügt werden, so dass gilt:

$$R = R' \cup \{F, T\} = \{F, T, A_1, \ldots, A_{d'}, K_1, \ldots, K_{d''}\} \qquad (4.16)$$

In den meisten Dateisystemen übt der Dateiname die Funktion eines Primärschlüssels aus, zumindest innerhalb eines Verzeichnisses. Liegen alle Dateinamen einschließlich des Pfades vor, gilt diese

Schlüsseleigenschaft über das gesamte Dateisystem. In diesem Fall kann das Attribut des Dateinamens direkt auf *F* abgebildet werden.

4.3.2 Semantische Dateisysteme

Imfeld stellt in **[Imf02]** fest, dass semantische Informationen zu Dateien, also Metadaten, grundsätzlich schon in den Dateien selbst vorhanden sind und prinzipiell auch extrahiert werden können. Da das Datenmodell *DS* üblicher Dateisysteme allen Dateien dasselbe Datei-Schema aufzwingt, ist der Zugang zu diesen Informationen einzig über die entsprechenden Applikationen möglich, wodurch die Sicht auf Metadaten eingeschränkt wird und ihre globale Verfügbarkeit verliert **[Imf02]**.

Semantische Dateisysteme stellen eine Weiterentwicklung physikalischer Dateisysteme zur Verbesserung dieser Situation dar. Ihr Schema *SDS* unterscheidet sich von dem traditioneller Dateisysteme (*DS*) dadurch, dass für alle Dateiformate individuelle Datei-Schemata und damit Attribute bzw. Metadaten verwaltet werden können:

$$SDS = \{R'_1, \ldots, R'_{k'}\} \quad (4.17)$$

Oft sind semantische Dateisysteme so konzipiert, dass sie auf ein traditionelles Dateisystem aufbauen, und ihr Datenmodell über eine zusätzliche Schnittstelle erreichbar ist **[OSC06]**. Über diesen Weg können beliebige Anwendungsprogramme auf die Metadaten zugreifen, die aufgrund der Beschränkungen physikalischer Dateisysteme innerhalb eines Dateikörpers gespeichert werden müssen. Sie werden von semantischen Dateisystemen gleichberechtigt ins Datei-Schema integriert:

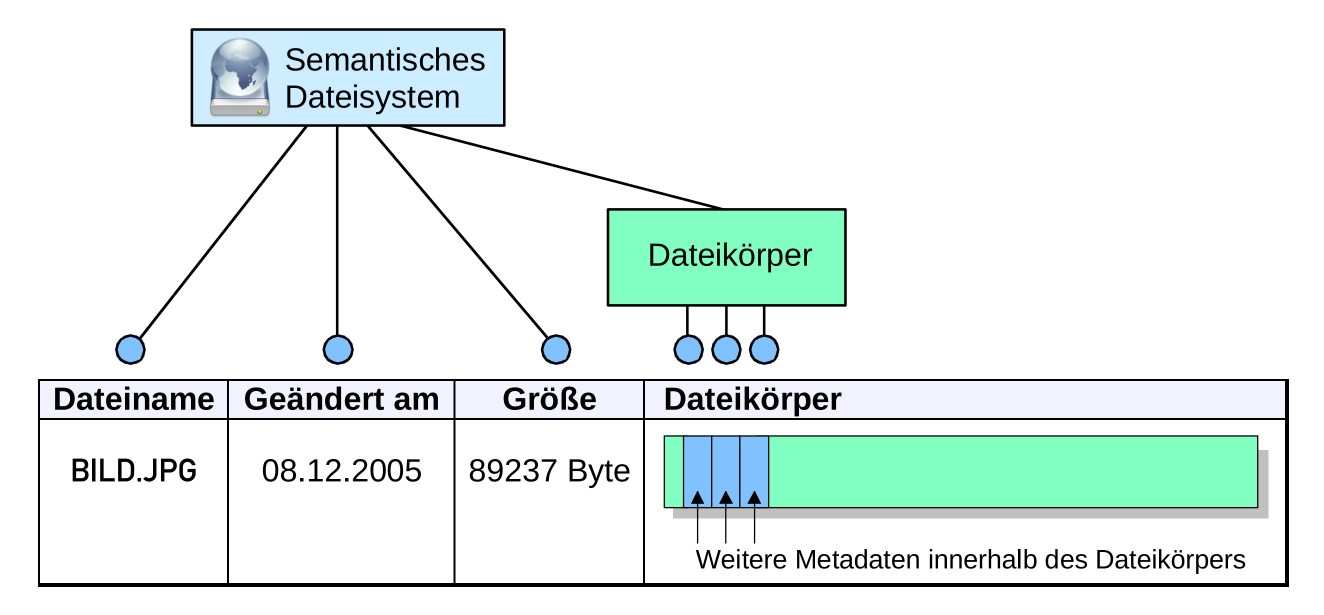


Abb. 4.3-2: Beispieldatei eines semantischen Dateisystems

Durch Abbildung aller R'_i aus *SDS* lässt sich analog zu (4.16) das Datenmodell eines semantischen Dateisystems in das einer Library überführen, ohne dass Informationen verloren gehen.

Für semantische Dateisysteme ist auch der umgekehrte Weg möglich: da sowohl ein Library-Schema als auch das Schema eines semantischen Dateisystems aus **mehreren** Objekt- bzw. Datei-Schemata bestehen, kann eine Library in ein semantisches Dateisystem überführt werden, wenn dies auch für jedes Objekt-Schema gilt.

Alle Attribute A_i eines Objekt-Schemas R mit $dom(A_i) = BLOB$ werden auf Dateikörper K_i eines Datei-Schemas R' abgebildet. Alle anderen Attribute werden auf Metadaten-Attribute des Datei-Schemas abgebildet, die ggf. transparent in einem der Dateikörper gespeichert werden. Das Primärschlüssel-Attribut F eines Objekt-Schemas kann, beispielsweise durch Umformung und Einbeziehung des Typ-Attributs T in Form eines Namenssuffix, auf den Dateinamen abgebildet werden.

Eine solche Abbildungsvorschrift kann im allgemeinen Fall nur ein Element eines semantischen Dateisystems *SDS* zum Ziel haben, da das uniforme Datei-Schema physikalischer Dateisysteme *DS* höchstens einem Objekt-Schema entsprechen kann. Damit gilt analog zu → *Kap. 4.2*, dass sich die Datenmodelle von physikalischen und semantischen Dateisystemen in ein Library-Schema überführen lassen, das Datenmodell einer Library sich hingegen nur auf ein semantisches Dateisystem abbilden lässt:

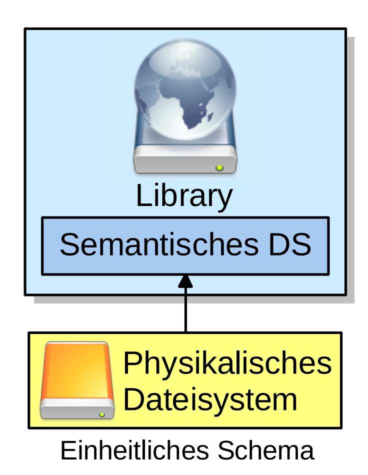


Abb. 4.3-3: Physikalisches Dateisystem, semantisches Dateisystem und Library

4.4 Vorteile

Marsden stellt in **[Mar03]** fest, dass Datenbanken ursprünglich in hierarchielosen Dateien gespeichert wurden. Da so keine Beziehungen zwischen den gespeicherten Daten modelliert werden konnten, wurden hierarchische Datenbanken eingeführt, die die Darstellung von 1:n-Beziehungen erlaubten. Hierarchische Datenbanken sind jedoch nicht geeignet, um alle Beziehungen zwischen Entitäten zu modellieren, weshalb die Hierarchie mittels Links durchbrochen werden konnte. An genau diesem Entwicklungspunkt befinden sich Dateisysteme heute **[Mar03]**.

Die Datenbanktechnik hat sich jedoch zu relationalen Datenbanken **[Cod70]** weiterentwickelt, die seit Anfang der 1990er-Jahre einen Siegeszug angetreten haben **[Saa05]**. Mit ihnen können Beziehungen dynamisch modelliert werden, und zwar ohne eine externe Hierarchie **[Mar03]**. Dies wird auch beispielsweise von Microsoft im Rahmen einer »data platform vision« angestrebt **[Mic07]**.
Durch das Ergänzen des relationalen Datenmodells um BLOBs und das Verwalten typspezifischer Datei-Schemata in Dateisystemen können beide Datenmodelle auf eine Library, und dadurch auch auf das jeweils andere Datenmodell, abgebildet werden. BLOB-relationale Datenbanken und semantische Dateisysteme sind also gleich mächtig, und in diesem Sinne als äquivalent anzusehen:

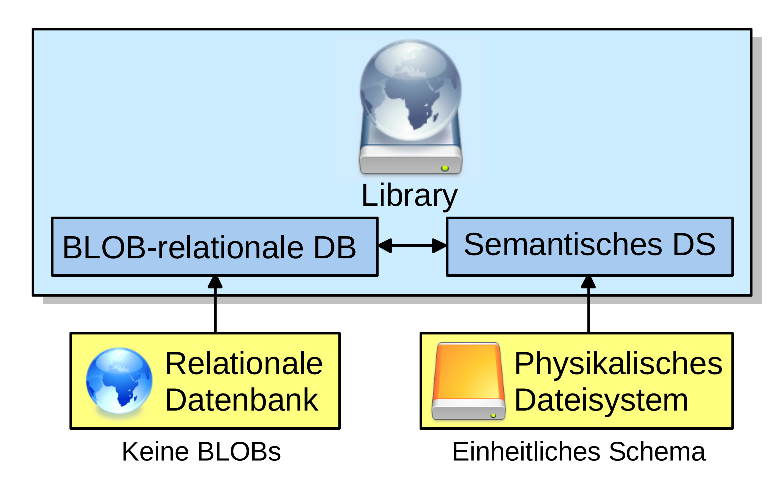


Abb. 4.4-1: Beziehung zwischen Datenbanken und Dateisystemen

4.4.1 Zugriff

In der Praxis ergeben sich aus den äquivalenten Datenmodellen erhebliche Vorteile, die u.a. zur Lösung der von Shoens et al. in **[Sho93]** gefundenen Problematik beitragen.

Das in → *Abb. 4-1* gezeigte EMail-Postfach wird physikalisch in einer proprietären Datenbank des zugehörigen Programms gespeichert, und ist daher für die Suchfunktionen des Betriebssystems nicht erreichbar. Es handelt sich bei der Datei EBAY.MBX um eine BLOB-relationale Datenbank: jedes Tupel enthält Attribute wie Sender, Empfänger, Empfangsdatum und Titel der Nachricht. Ein BLOB enthält den Textkörper, ggf. werden weitere BLOBs für angehängte Dateien hinzugefügt.

Betriebssysteme, die die Integration von Datenbanken und Dateisystemen zu einer umfassenden Library umsetzen, können die proprietäre Datenbank des EMail-Programms mounten, also durch ein geeignetes Schema-Mapping (→ *Kap. 4.2*) in die Library einbinden. Die einzelnen Tupel werden dabei gleichberechtigt mit Dateien als hochdimensionale Datenobjekte aufgefasst, und sind für die Dateisuche zugänglich. Im Beispiel würde also nicht die physikalische Datei EBAY.MBX zum Suchergebnis hinzugefügt, sondern individuelle EMail-Objekte, die das Suchkriterium erfüllen. Die enthaltenen Daten werden dadurch für das Betriebssystem und andere Applikationen zugänglich.

4.4.2 Operationen

Die Äquivalenz der Datenmodelle impliziert, dass auf der Management-Ebene (→ *Abb. 4-2*) beide Konzepte auf dieselbe Weise genutzt werden können, insbesondere ein Technologieaustausch möglich ist: so können beispielsweise Beziehungen zwischen Datenobjekten durch die enthaltenen Metadaten statt durch Verzeichnisse modelliert werden, und Suchvorgänge bzw. Veränderungen durch eine grafische Oberfläche (→ *Kap. 7*) statt einer Abfragesprache durchgeführt werden. Ähnlich wie relationale Datenbanken bei ihrer Einführung 1:*n*-Beziehungen zu *n*:*m*-Beziehungen erweitert haben, so implizieren Libraries das Durchbrechen einer starren, durch Verzeichnisse vorgegebenen 1:*n*-Hierarchie **[Mar03]**.

Darüber hinaus lassen sich viele alltägliche Operationen, die physikalische Dateisysteme durchführen können, auf die Relationenalgebra zurückführen, da diese auch auf das Library-Modell anwendbar ist. So entspricht das Ausgeben aller Dateien in einem bestimmten Verzeichnis und mit einem bestimmten Namen, etwa durch DIR ∗.DOC bzw. LS ∗.DOC, einer Selektion σ:

```
ROOT@LocalHost|E:\>dir *.doc

Die Bezeichnung von Laufwerk E: ist EXTENSION
Seriennummer: 401A/180D
Suchmaske: E:\*.DOC

MPGSYS   DOC        286.208  -------  13.09.2006 10:09:54
MPGVIDEO DOC        642.560  -------  13.09.2006 10:09:48

                928.768 Byte in 2 Dateien und 0 Verzeichnissen
        108.829.114.368 Byte frei

ROOT@LocalHost|E:\>
```

Abb. 4.4-2: Selektion in einem physikalischen Dateisystem

Je nach eingesetztem Betriebssystem können durch zusätzliche Parameter unterschiedliche Attribute für jede Datei ausgegeben werden. In diesem Beispiel gibt /B nur die Dateinamen ohne weitere Angaben aus; es handelt sich dabei also um eine Selektion σ mit anschließender Projektion π auf dieses eine Attribut:

```
ROOT@LocalHost|E:\>dir *.doc /B

MPGSYS.DOC
MPGVIDEO.DOC

ROOT@LocalHost|E:\>
```

Abb. 4.4-3: Selektion und Projektion in einem physikalischen Dateisystem

4.4.3 Typsicherheit

Traditionell identifizieren Betriebssysteme und Applikationen den Typ einer Datei durch eine Erweiterung des Dateinamens, wie etwa .avi, .c, .jpg, .mp3 und so weiter. Dieses Verfahren hat den

großen Nachteil, dass ein Anwender beim Umbenennen einer Datei die Erweiterung und damit den Dateityp ändern kann, was die Datei unbrauchbar macht (sie würde z.B. beim Öffnen gar nicht oder mit einem falschen Programm gestartet):



Abb. 4.4-4: Warnung beim Verändern der Dateinamenerweiterung

Im Gegensatz zu traditionellen Dateisystemen enthalten alle Objekt-Schemata einer Library mit T eine eindeutige Beschreibung für das Format (4.6). Da alle Datenobjekte eines Schemas R denselben Attributwert t enthalten, fungiert dieser als eindeutiger Typbezeichner des Schemas selbst – etwa als Index von R in der Library L (4.7). Analog zu XML-Dateien, die ihr Schema referenzieren **[Mic08d]**, müssen jedoch auch alle Datenobjekte einer Library ihr Objekt-Schema eindeutig referenzieren, weshalb der Typbezeichner als Attribut T in R modelliert wurde. Dadurch wird eine Library inhärent typsicher, so dass Klassifizierungs-Module (→ *Kap. 2.3.1*), die den Dateityp anhand einer Heuristik ermitteln **[Sho93]**, überflüssig werden. Da auch der Dateiname bzw. seine Endung nicht mehr für die Codierung des Dateiformats benötigt wird, kann eine Datei problemlos beliebig umbenannt werden.

4.4.4 Terminologie

Eine weitere Konsequenz, die sich aus den äquivalenten Datenmodellen ergibt, ist eine vereinheitlichte Terminologie für Speichersysteme. Die Tabelle in → *Abb. 4.4-5* stellt zeilenweise die äquivalenten Begriffe von relationalen Datenbanken, Libraries und Dateisystemen dar, und vergleicht diese zusätzlich mit der proprietären Nomenklatur von Microsoft WinFS (→ *Kap. 2.9*):

Datenbanken	Libraries	Dateisysteme	MS WinFS (→ *Kap 2.9*)
BLOB	–	Dateikörper	–
Tupel	Datenobjekt	Datei	Item
Schema	Schema	Dateiformat, -typ	Schema
Datenbank	Library	Dateisystem	Store

Grün hervorgehoben: im weiteren Verlauf eingesetzte Begriffe

Abb. 4.4-5: Terminologie für Datenbanken, Libraries und Dateisysteme

Shoens bemerkt in **[Sho93]**, dass sich die Anwender schon vor langer Zeit für Dateisysteme und gegen Datenbanken zur Speicherung von Informationen entschieden haben. Über 15 Jahre später ist diese Aussage noch immer wahr, und betrifft aufgrund der großen Verbreitung digitaler Medien heute weitaus mehr Menschen als 1993.

Zur besseren Lesbarkeit werden daher im Verlauf dieser Arbeit die in → *Abb. 4.4-5* grün hervorgehobenen Begriffe verwendet. Eine Ausnahme stellen formale Betrachtungen oder der explizite Bezug auf Datenbanken bzw. Dateisysteme dar. Der Begriff »Library« wird bereits informell in diversen Produkten, etwa im Windows Media Player oder in Windows 7, verwendet:



Abb. 4.4-6: Verwendung des Begriffs »Library« im Microsoft Windows Media Player 11

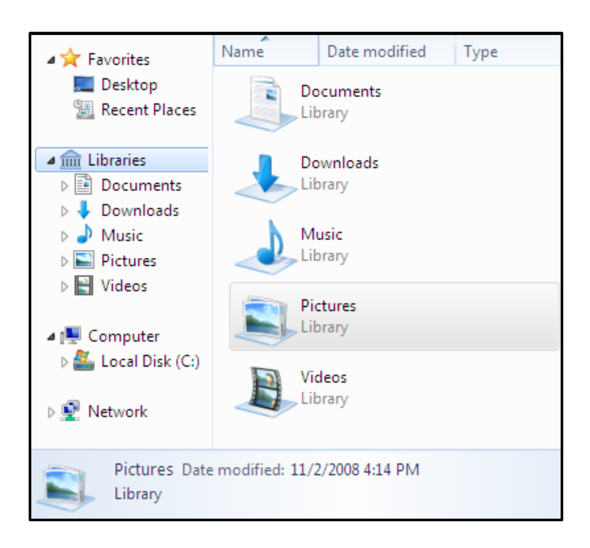


Abb. 4.4-7: Verwendung des Begriffs »Library« in Microsoft Windows 7

5 Indexierung

In diesem Kapitel wird eine neuartige Indexstruktur, der »Master/Slave-Index« **[Kol07a] [Kol08c]**, eingeführt. Diese Datenstruktur kann insbesondere eingesetzt werden, um die Attribute von Dateien zu indexieren und eine hohe Performanz für die Dateisuche (→ *Kap. 6.2.2*) sicherzustellen. Der Master/Slave-Index wird in einer Referenz-Library (→ *Kap. 6*) eingesetzt, die gleichzeitig der Durchführung praktischer Performanzmessungen dient (→ *Kap. 6.3*).

5.1 Master/Slave-Index

Die Grundidee des Master/Slave-Index besteht darin, die Attributwerte aller Dateien hintereinander in einer Heap-Datei **[Saa05]** zu speichern. Suchanfragen werden bearbeitet, indem diese Datei sequenziell gelesen wird, und die Attributwerte jeder Datei mit einem Suchfilter (→ *Kap. 6.2.2*) abgeglichen werden.

Diese Vorgehensweise ist immun gegen den »Fluch der Dimensionen« (→ *Kap. 3.3.1*), denn die Laufzeit einer Indexabfrage ist ausschließlich von der Indexgröße und der Geschwindigkeit des Datenträgers abhängig. Dadurch ist die Performanz auch unabhängig von der Anzahl der Dimensionen des Datenraums, denn das Einlesen einer Zeichenkette mit 256 Byte Länge hat denselben Aufwand wie das Einlesen von 256 8-Bit-Zahlen. Außerdem werden Partial Match Abfragen, bei denen nur wenige Attribute spezifiziert sind (→ *Kap. 3.3.2*), problemlos unterstützt.

Das sequenzielle Lesen der Indexdatei hat darüber hinaus den Vorteil, dass kein zusätzlicher Leistungsverlust durch das eingesetzte Dateisystem auftritt (→ *Kap. 3.3.3.2*). Auf diese Weise kann ein Master/Slave-Index problemlos innerhalb eines Dateisystems abgespeichert werden. Da bei unfragmentierten Dateien große Teile in zusammenhängenden Blöcken auf dem Datenträger gespeichert sind, können diese Bereiche sequenziell gelesen und im Arbeitsspeicher gepuffert werden. Somit werden Verzögerungen durch langsamen wahlfreien Zugriff (→ *Kap. 3.3.3.1*) vermieden.

5.1.1 Generalisierung/Spezialisierung

Durch die unterschiedlichen Formate der einzelnen Dateien entsteht ein partiell belegter Datenraum (→ *Kap. 3.3.4*). Um heterogene Tupel in einer Heap-Datei abzuspeichern, gibt es prinzipiell zwei Möglichkeiten: entweder müssen unterschiedliche Tupelgrößen in Kauf genommen werden, die die Implementierung erschweren und ineffizienter werden lassen, da kein wahlfreier Zugriff anhand der Tupelnummer mehr möglich ist, oder jedes Tupel wird auf eine Maximalgröße verlängert, wodurch die Gesamtgröße des Indexes zunimmt. Beide Varianten sind nicht akzeptabel, zumal letztere die Einführung neuer Dateiformate mit weiteren Attributen erschwert, wenn nicht ausreichend Platz in den Tupeln reserviert wurde:

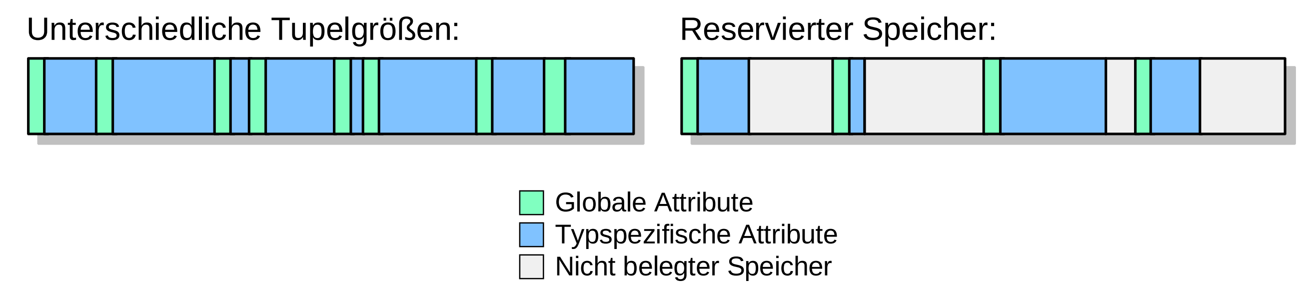


Abb. 5.1-1: Suboptimale Datenstrukturen

Daher wird der Master/Slave-Index als eine zweistufige Hierarchie implementiert, die eine Generalisierung bzw. Spezialisierung abbildet. Ein Master-Index speichert alle Attribute, die in allen unterstützten Dateiformaten vorkommen (z.B. Primärschlüssel F, Typ T, Dateiname, Größe, Zeitpunkt der letzten Änderung, Zugriffsrechte). Dem Master-Index werden zusätzliche Slave-Indexe für typspezifische Attribute zur Seite gestellt, wodurch der partiell belegte Datenraum (→ *Kap. 3.3.4*) abgebildet wird **[Kol08c]**.

Die formale Definition der Struktur eines Master/Slave-Index basiert auf der Definition einer Library (→ *Kap. 4.1*). Damit nicht die Dateien aller Formate indexiert werden müssen (→ *Kap. 6.1.1*), wird zunächst basierend auf einem Library-Schema L ein Index-Schema L' definiert:

$$L'=\{R'_1,\ldots,R'_{k'}\}\subseteq L,\ k'\leq k \tag{5.1}$$

Da nur ausgewählte Attribute aus $glob(L)$ indexiert werden sollen (also beispielsweise große Dateikörper vom Index ignoriert werden sollen), wird jedes R'_j aus L' als Teilmenge eines $R_{j'}$ aus L definiert. Diese Teilmenge muss jedoch das Primärschlüsselattribut $R_{j'}.F$ und das Typ-Attribut $R_{j'}.T$ enthalten:

$$\forall R'_j\in L' : R'_j\subseteq R_{j'},\ R_{j'}\in L,\ R_{j'}.F \wedge R_{j'}.T\in R'_j \tag{5.2}$$

Der Datenraum $\Omega(L')$ und der Objektraum $\omega(L')$ weisen dieselbe Struktur wie $\Omega(L)$ bzw. $\omega(L)$ auf, im Vergleich fehlen allerdings als unwichtig angesehene Dateiformate und Attribute. Analog zu (4.9) seien als X_i die n' Attribute aus $common(L')$ bezeichnet, welche in allen k' Schemata R'_j vertreten sind – nach Bedingung (5.2) also insbesondere F und T. $Y_{j,i}$ seien die m'_j übrigen Attribute aus R'_j. Dann gilt:

$$\omega(L')=\bigcup_{j=1}^{k'} dom(X_1)\times\ldots\times dom(X_{n'})\times dom(Y_{j,1})\times\ldots\times dom(Y_{j,m'_j}),\quad \begin{matrix} X_i\in common(L')\wedge \\ Y_{j,i}\in R'_j-common(L') \end{matrix} \tag{5.3}$$

Ein vollständiger Master/Slave-Index ist eine Menge von Relationen, die aus genau einer ausgezeichneten Relation (Master-Index m) und einer Relation für jedes der k' zu indexierenden Dateiformate besteht (Slave-Indexe s_j):

$$Index = \{m, s_1, \ldots, s_{k'}\} \tag{5.4}$$

Der Master-Index m speichert die Werte aller n' typunabhängigen Attribute X_1 bis $X_{n'}$. Die in m enthaltenen Tupel sind das Ergebnis einer Selektion σ aller Datenobjekte der zu indexierenden Dateiformate, projiziert auf die Attribute X_1 bis $X_{n'}$ (π und σ bezeichnen die Projektion bzw. Selektion der Relationenalgebra):

$$m \subseteq \pi_{X_1,\ldots,X_{n'}}(\Omega(L')) = dom(X_1) \times \ldots \times dom(X_{n'})$$
$$m = \bigcup_{j=1}^{k'} \pi_{X_1,\ldots,X_{n'}}\left(\sigma_{t=Typbez.\ \text{für}\ R'_j}(l(L))\right) \tag{5.5}$$

Ein Slave-Index s_j speichert ebenfalls den Schlüssel aller jeweils indexierten Dateien ($R'_j.F$) sowie alle m'_j Attribute $Y_{j,i}$, die für das jeweilige Objekt-Schema R'_j spezifisch sind und indexiert werden. Die Tupel in s_j sind ebenfalls das Ergebnis einer Selektion in $l(L)$ mit anschließender Projektion:

$$s_j \subseteq \pi_{F,Y_{j,1},\ldots,Y_{j,m'_j}}(\Omega(L')) = dom(F) \times dom(Y_{j,1}) \times \ldots \times dom(Y_{j,m'_j})$$
$$s_j = \pi_{F,Y_1,\ldots,Y_{j,m'_j}}\left(\sigma_{t=Typbez.\ \text{für}\ R'_j}(l(L))\right) \tag{5.6}$$

5.1.2 Retrieval

Suchanfragen können als »Natural Join« **[Saa05]** zwischen Master- und Slave-Indexen durchgeführt werden, also $m \bowtie s_1$ bis $m \bowtie s_{k'}$. Die Zahl der beteiligten Slave-Indexe ist aus Performanzgründen zu minimieren (»*Query containment*«, → *Kap. 6.2.2.1*), so dass beispielsweise bei Abfragen über Dateien eines bestimmten Typs j nur $m \bowtie s_j$ berechnet werden muss.

Da nun keiner der Indexe seine Elemente anhand des Primärschlüssels F sortiert, erscheint es bei naiver Betrachtung so, als müsste die Join-Operation durch »Nested Loops« in $O(n^2)$ implementiert werden. Wären die Slave-Indexe hingegen bezüglich des Primärschlüssels F sortiert, könnte für jeden Eintrag im Master-Index eine binäre Suche durchgeführt werden, so dass sich die Laufzeit auf $O(n \log n)$ verbessern würde.

Eine binäre Suche ist jedoch gar nicht erforderlich, denn tatsächlich kann beim Master/Slave-Index der Join als »Merge-Join« **[Saa05]** in $O(n)$ berechnet werden, indem eine zusätzliche Eigenschaft eingeführt wird: für zwei beliebige Dateien des selben Typs gilt, dass ihre relative Position zueinander im Master-Index m und im zugehörigen Slave-Index s_j stets dieselbe sein muss **[Kol08c]**. Dies kann auch formalisiert werden:

$$\forall o_1, o_2 \in m : (\pi_T(o_1) = \pi_T(o_2)) \wedge Pos(\pi_F(o_1), m) < Pos(\pi_F(o_2), m) \Rightarrow Pos(\pi_F(o_1), s_j) < Pos(\pi_F(o_2), s_j)$$
$$Pos(f, r) = \text{Zeilenposition des Tupels mit dem Primärschlüsselwert } f \text{ in der Relation } r \tag{5.7}$$

Ist diese Eigenschaft erfüllt, so können alle Index-Relationen parallel sequenziell durchlaufen werden, was in $O(n)$ möglich ist und einem Merge-Join **[Saa05]** entspricht.

5.1.2.1 Beispiel

Das folgende Beispiel zeigt einen Master/Slave-Index, der die Attributwerte einiger Bilder und MP3-Dateien enthält. Das Attribut F wird als »Filekey« bezeichnet, das Attribut T als »Typ«. Darüber hinaus besitzt der Beispiel-Index die oben definierte Merge-Eigenschaft (5.7):



Master-Index

Filekey	Typ	...
Z6_HG7E$	Bild	...
JK83ZZA_	MP3	...
LL9)ACX1	Bild	...
JGUIP7$D	MP3	...
30_KUIXY	Bild	...



Slave-Index für Bilder

Filekey	Breite	Höhe
Z6_HG7E$	1024	768
LL9)ACX1	640	480
30_KUIXY	2048	1536

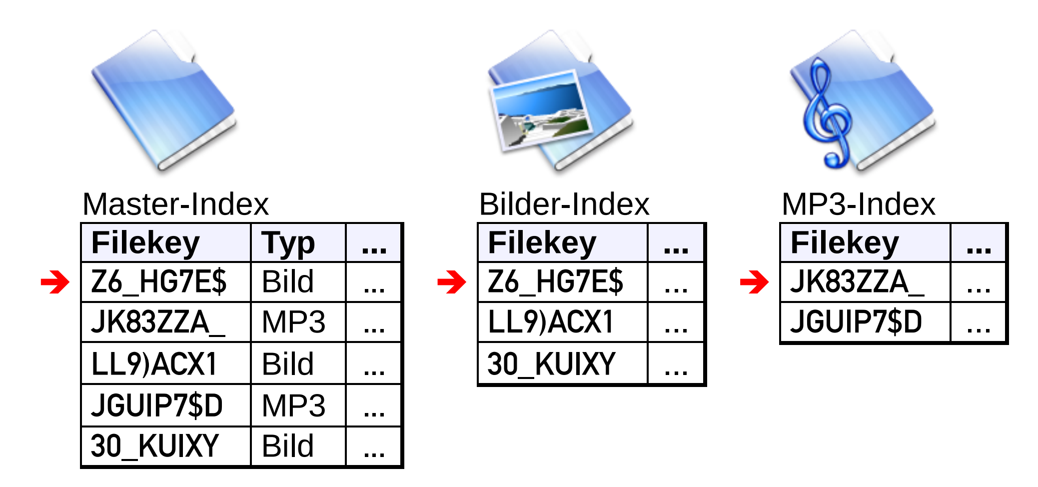


Slave-Index für MP3s

Filekey	Künstler	Album
JK83ZZA_	Texas	The Hush
JGUIP7$D	Hacienda	3rd door

Abb. 5.1-2: Master/Slave-Index **[Kol08c]**

Zur Bearbeitung von Suchanfragen wird jeder Index mit einem Zeiger auf das nächste zu bearbeitende Element versehen. Anfangs zeigen alle Merker auf das erste Element ihres Index:



Master-Index

	Filekey	Typ	...
→	Z6_HG7E$	Bild	...
	JK83ZZA_	MP3	...
	LL9)ACX1	Bild	...
	JGUIP7$D	MP3	...
	30_KUIXY	Bild	...

Bilder-Index

	Filekey	...
→	Z6_HG7E$	...
	LL9)ACX1	...
	30_KUIXY	...

MP3-Index

	Filekey	...
→	JK83ZZA_	...
	JGUIP7$D	...

Abb. 5.1-3: Startsituation **[Kol08c]**

Die erste zu untersuchende Datei im Beispiel ist die Bilddatei mit dem Schlüssel Z6_HG7E$. Die typunabhängigen Attribute stehen sofort durch Abfrage des Master-Index zur Verfügung. Für typspezifische Attribute muss nun der entsprechende Eintrag im Index für Bild-Metadaten »gesucht«

werden. Dieses »Suchen« ist allerdings ein einfaches »Finden« in $O(1)$, da der Merker im Bilder-Index bereits auf den gewünschten Eintrag zeigt. Danach werden die Zeiger im Master- und Bilder-Index um eine Position weiterbewegt:

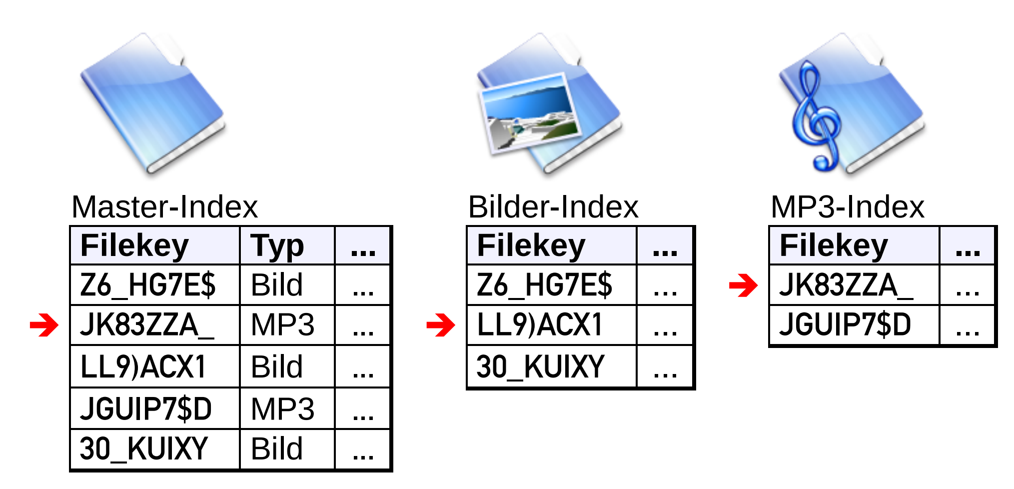


Abb. 5.1-4: Nach Bearbeitung der ersten Datei **[Kol08c]**

Analog zur ersten Datei wird nun die MP3-Datei mit dem Schlüssel **JK83ZZA_** bearbeitet, die sich im Master-Index an der zweiten Stelle befindet. Der Merker im MP3-Index zeigt sofort auf den korrekten Eintrag, so dass in $O(1)$ auf die typspezifischen Metadaten zugegriffen werden kann. Danach werden wieder alle betroffenen Zeiger, also diejenigen im Master und MP3-Index, auf das jeweils nächste Element bewegt:

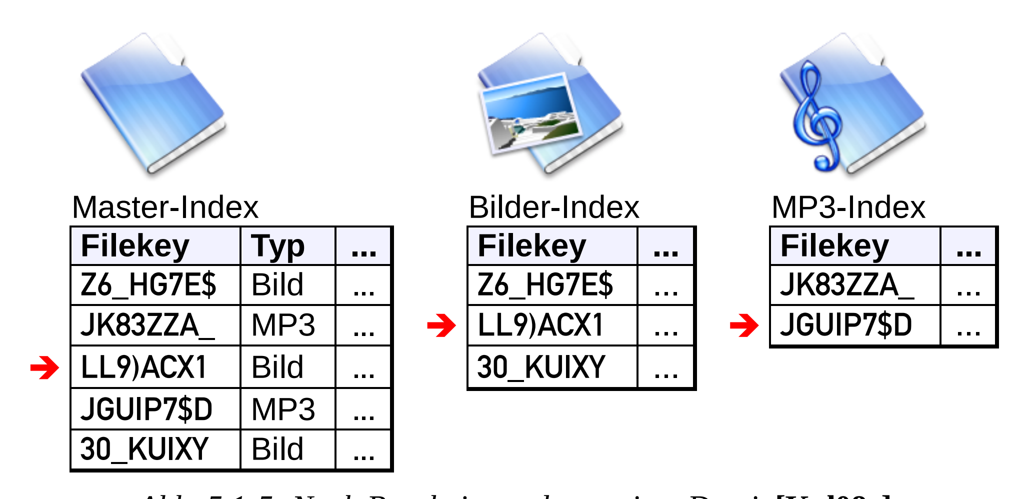


Abb. 5.1-5: Nach Bearbeitung der zweiten Datei **[Kol08c]**

Die dritte Datei, die der Master-Index liefert, ist wieder ein Bild. Die Metadaten für das Bild mit dem Schlüssel **LL9)ACX1** finden sich wieder an der Stelle, auf die der Merker im Bilder-Index zeigt, und so weiter.

Damit dieser Merge-Join funktioniert, müssen alle Indexe die Merge-Eigenschaft (5.7) erfüllen. Die übrigen Operationen müssen dazu auf Algorithmen basieren, die dies stets gewährleisten **[Kol08c]**.

5.1.3 Hinzufügen

Sollen neue Dateien zum Index hinzugefügt werden, so müssen die entsprechenden Tupel ans Ende der Index-Relationen angehängt werden. Dadurch wird die Sortierung der bereits vorhandenen Tupel nicht geändert. Die neuen Tupel haben automatisch die höchste Position in den jeweiligen Relationen, so dass auch für sie die Merge-Eigenschaft gilt.

5.1.4 Ändern

Zum Ändern eines Tupels wird der Index wie beim Retrieval (→ *Kap. 5.1.2*) sequenziell durchlaufen. Wird das zu ändernde Tupel anhand des Schlüssels gefunden, so wird es aktualisiert. Dieser Vorgang kann die Reihenfolge von Tupeln nicht beeinflussen, da die Position des Tupels innerhalb einer Index-Relation selbst bei einer Änderung des Filekeys unverändert bleibt.

5.1.5 Löschen

Wird eine Datei gelöscht, muss das entsprechende Tupel aus dem Master-Index und dem betroffenen Slave-Index entfernt werden. Das Löschen von Tupeln ist potenziell gefährlich – bei naiver Implementierung kann die Sortierung zerstört werden, wie im folgenden Abschnitt demonstriert wird.

5.1.5.1 Löschen von Dateien (naiv)

Aufgrund der uniformen Tupelgröße in den jeweiligen Indextabellen ist es naheliegend, das Löschen eines Tupels durch Überschreiben mit dem letzten Tupel und Verkürzen der Datei zu implementieren:

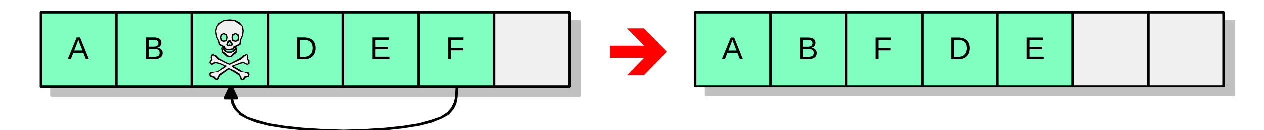
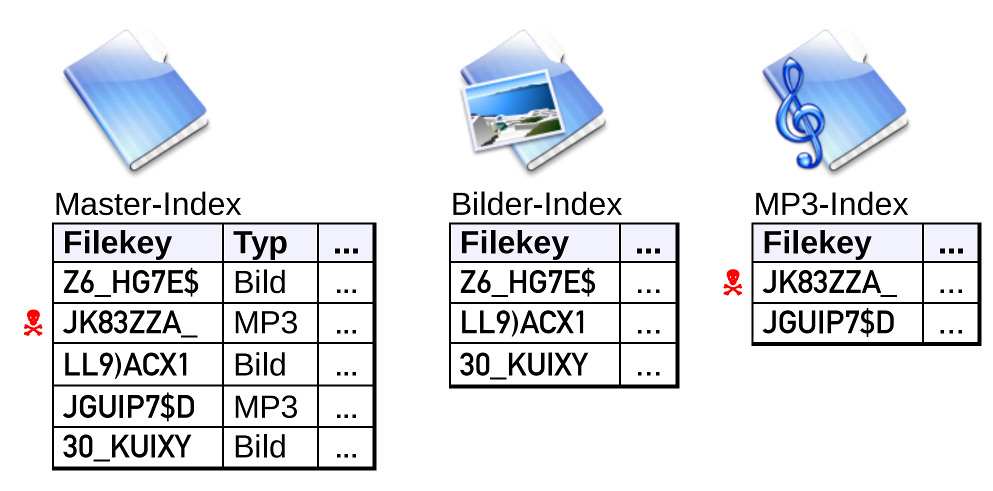


Abb. 5.1-6: Löschen eines Tupels durch Überschreiben mit dem letzten Tupel

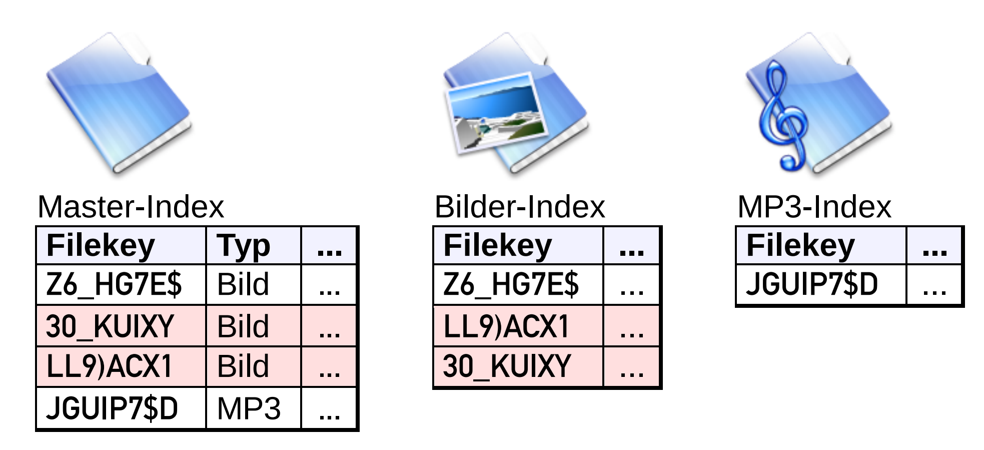
Das in diesem Abschnitt verwendete Beispiel liefert einen Fall, bei dem die Sortierung dadurch zerstört werden würde. Angenommen, die MP3-Datei mit dem Schlüssel **JK83ZZA_** soll gelöscht werden:



: zu löschende Einträge

Abb. 5.1-7: Vor dem Löschen

Die nachfolgende Abbildung zeigt dieselben Relationen nach dem Löschen der Datei; die betroffenen Einträge wurden absichtlich durch das jeweils letzte Tupel der Heap-Dateien überschrieben:



Rot hervorgehoben: ungleiche Reihenfolge

Abb. 5.1-8: Nach dem Löschen

Die in → *Abb. 5.1-8* rot hinterlegten Einträge befinden sich nun in ihren jeweiligen Relationen nicht mehr in derselben Reihenfolge zueinander, so dass ein Merge-Join die falschen Tupel verbinden würde. Im Master-Index hat der Eintrag mit dem Schlüssel **30_KUIXY** den Eintrag für **LL9)ACX1** »überholt«, was im Slave-Index nicht erfolgt ist – die Merge-Eigenschaft ist für diese beiden Tupel nicht mehr erfüllt.

5.1.5.2 Löschen von Dateien (korrekt)

Um die Merge-Eigenschaft zu erhalten, müssen Tupel durch Aufrücken der Nachfolger aus den Index-Relationen gelöscht werden:

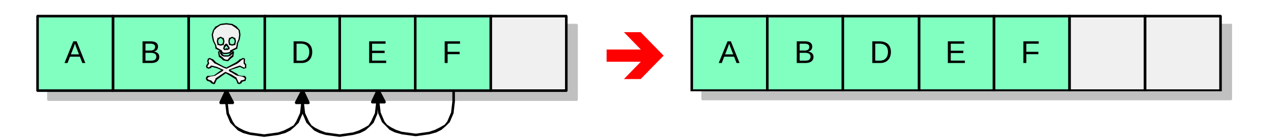


Abb. 5.1-9: Löschen eines Tupels durch Verschieben

Die Laufzeit einer korrekten Löschoperation beträgt immer $O(n)$, da alle nachfolgenden Tupel verschoben werden müssen.

5.1.6 Massen-Operationen

Der Master/Slave-Index hat bei Änderungs- und Lösch-Operationen lineare Laufzeit, was beim Einsatz eines effizienten Dateisystems wie z.B. ext2 eine partielle Verschlechterung bedeutet (→ *Abb. 5.2-2*). Das Verändern von Attributwerten und Löschen einer Datei ist in diesem Fall nicht mehr in $O(\log n)$, sondern nur noch in $O(n)$ möglich. Dieser Umstand fällt besonders auf, wenn sehr viele Dateien betroffen sind (sog. »batch rename« bzw. »batch delete«):

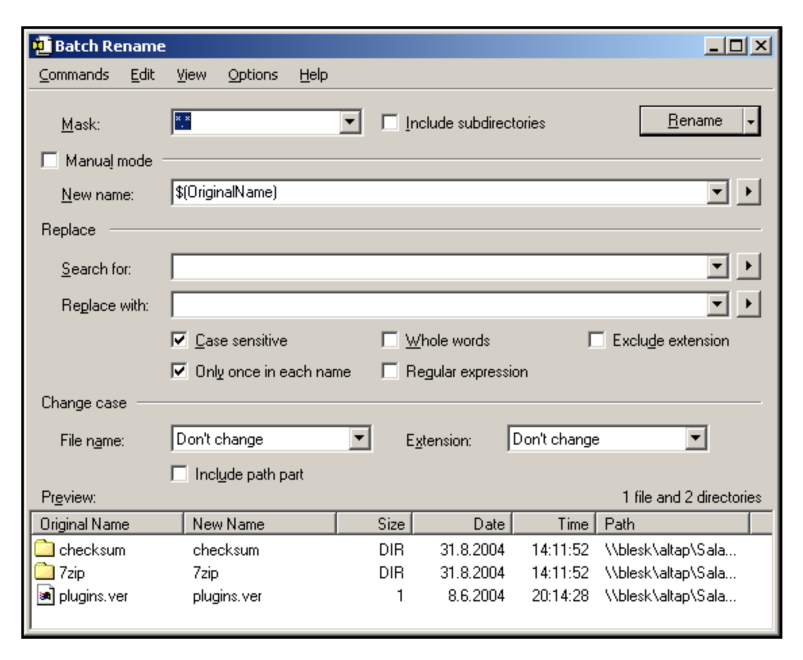


Abb. 5.1-10: »Batch rename«

Im Folgenden soll das Massenlöschen von Dateien näher untersucht werden. Der naive Ansatz, für alle m Dateien die Löschfunktion des Betriebssystems mit dem jeweiligen Schlüssel aufzurufen, ist von Nachteil. Viel effizienter ist es, dem Betriebssystem selbst eine Liste der zu löschenden Dateien zu übergeben, die dann direkt abgearbeitet wird.

Dass die zweite Variante effizienter ist, überrascht, denn in beiden Fällen ergibt sich theoretisch eine Laufzeit von $O(n*m)$. Dabei wird jedoch vernachlässigt, dass die Liste der zu löschenden Dateien zumeist im schnellen Arbeitsspeicher vorliegt, während das Dateisystem und der Index auf der deutlich langsameren Festplatte gespeichert sind.

Das Massenlöschen läuft im Index nun in zwei Phasen ab: in der ersten Phase werden ähnlich wie beim Suchen alle Einträge in den Indexen verarbeitet. Für jeden Eintrag wird in der Löschliste nach dem **Filekey** gesucht; wird er gefunden, werden alle Einträge der betroffenen Datei markiert:

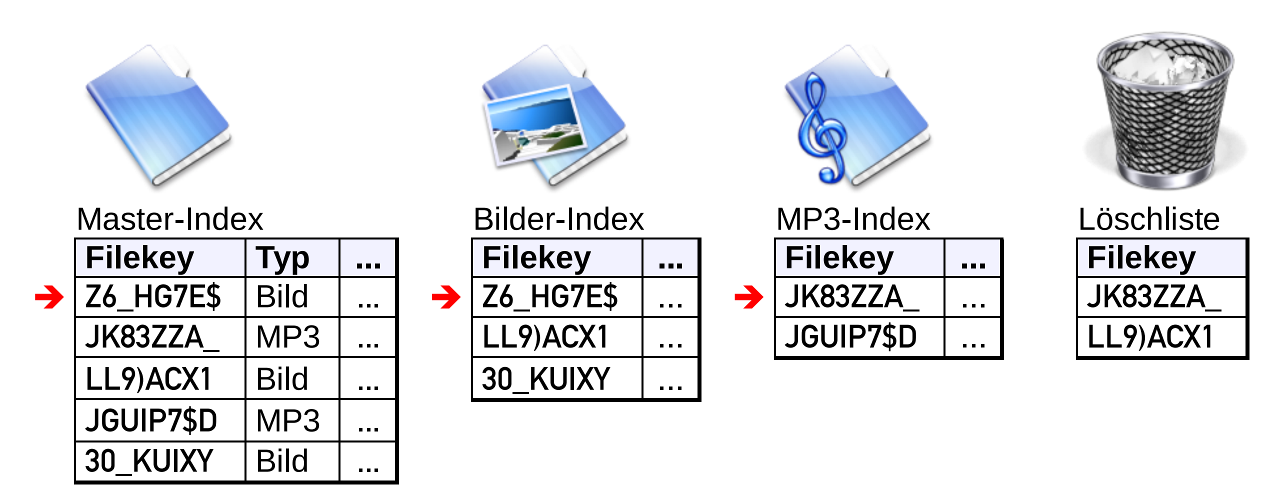


Abb. 5.1-11: Vor dem Massenlöschen

Das Löschen jener Dateien in der Löschliste entspricht weitgehend der bereits vorgestellten Dateisuche (→ *Kap. 5.1.2*). Die aktuelle Datei wird allerdings nicht auf das Erfüllen einer Suchbedingung geprüft oder an den Aufrufer zurückgeliefert, sondern in der Löschliste gesucht. Wird sie dort aufgefunden, werden ihre Einträge im Master/Slave-Index markiert, etwa durch Setzen eines Bits in einem speziellen Attribut oder durch Überschreiben des **Filekey** mit Nullbytes:

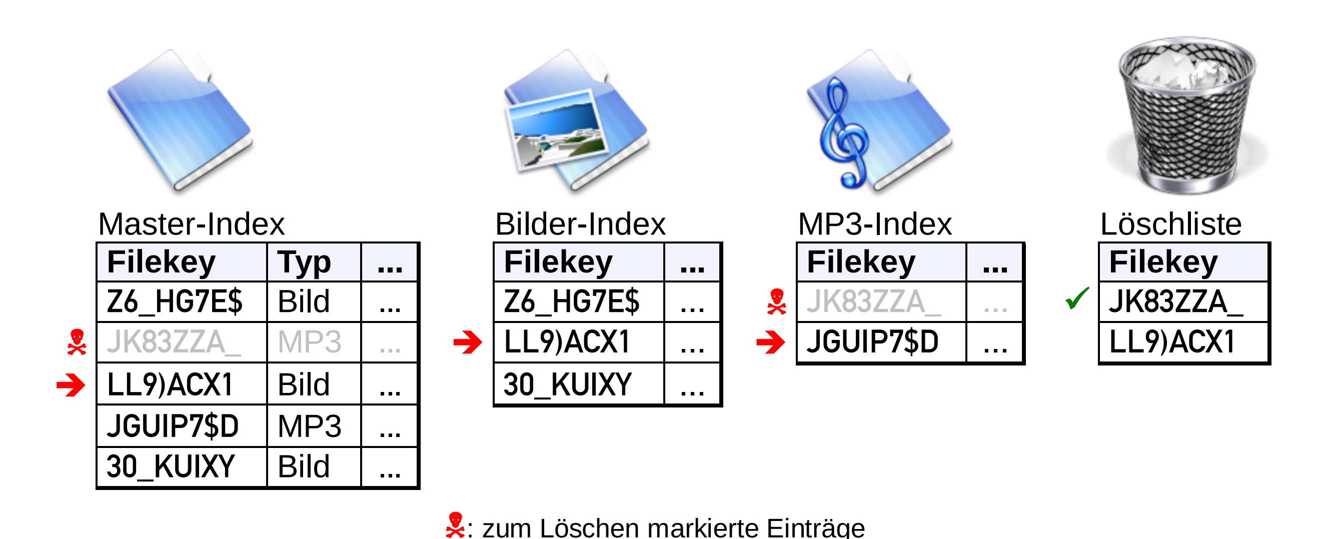


Abb. 5.1-12: Nach dem Markieren der ersten Datei

Da alle *n* Dateien aus dem Master-Index mit den *m* Einträgen in der Löschliste verglichen werden müssen, ergibt sich eine Ordnung von $O(n*m)$. Gegenüber dem Durchlaufen des Indexes, der auf der Festplatte gespeichert ist, fällt das Durchsuchen der Löschliste im Arbeitsspeicher allerdings kaum ins Gewicht. Liegt die Löschliste sortiert vor, kann beim Suchen der aktuellen Datei sogar eine binäre Suche eingesetzt werden, so dass sich die Ordnung zu $O(n \log m)$ verbessert. Dies ist bei der naiven Variante nicht möglich, da die Sortierung der Löschliste dort irrelevant ist. Nach der ersten Phase sind alle Dateien der Löschliste aufgefunden und die zugehörigen Einträge in allen Indexen markiert:

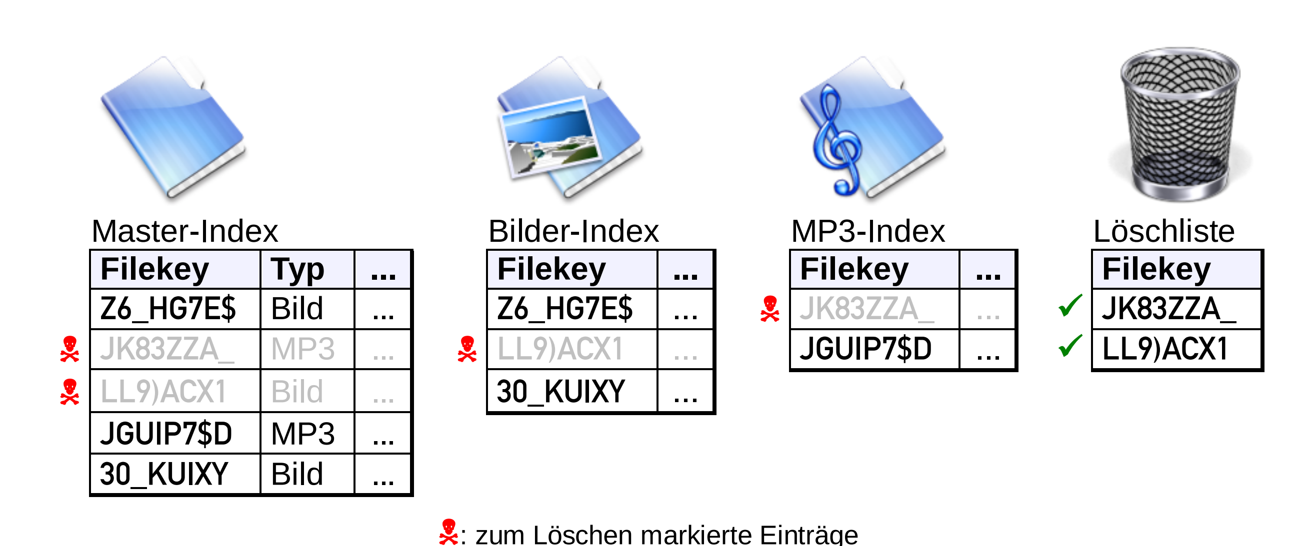


Abb. 5.1-13: Nach dem Markieren aller Dateien der Löschliste

In einer zweiten Phase müssen nun alle markierten bzw. invalidierten Datensätze endgültig aus der Datei entfernt werden. Dieser Schritt muss jedoch nicht sofort durchgeführt werden, sondern kann auch beispielsweise bis zum Logout des Benutzers verschoben werden. Das Entfernen der Datensätze hat pro Tabelle eine Laufzeit von $O(n)$, wenn jeder Index sequenziell verarbeitet wird und nicht markierte Einträge an eine neue Heap-Datei angehängt werden. Diese neue Datei ersetzt schließlich die ursprüngliche Datei, so dass alle zu löschenden Tupel endgültig aus dem Index entfernt sind.

5.2 Performanz

Die Tabelle in → *Abb. 5.2-2* stellt für die Dateisysteme FAT und ext2 die Dauer verschiedener Operationen ohne und mit Master/Slave-Index dar. Das FAT-Dateisystem speichert Verzeichniseinträge in einer Liste ab, während ext2 dafür einen B^+-Baum einsetzt. Das Suchen einer bestimmten Datei anhand ihres Dateinamens ist also in $O(n)$ bzw. $O(\log n)$ durchführbar.

Wenn eine Liste mit allen Dateien vorliegt, ist eine auf Metadaten gestützte Suche beim Einsatz des FAT-Dateisystems in $O(n^2)$ möglich: jede der *n* Dateien muss geöffnet und durchsucht werden, wo-

bei das Öffnen jeweils $O(n)$ zum Suchen des Verzeichniseintrags benötigt. Für ext2 ergibt sich entsprechend $O(n \log n)$:

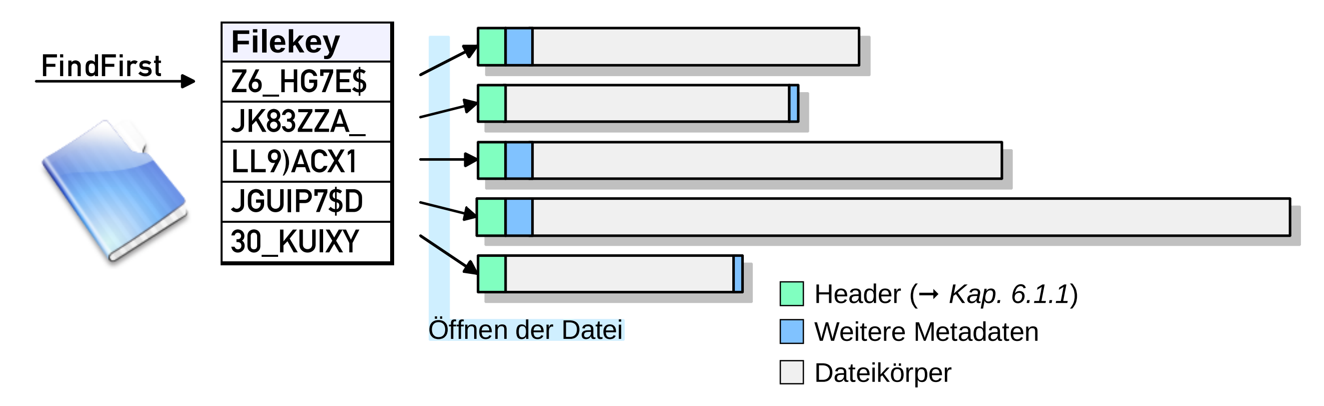


Abb. 5.2-1: Ablauf eines Suchvorgangs ohne Index

Das Modifizieren eines Verzeichnisses (Hinzufügen oder Löschen von Dateien bzw. Modifizieren des Verzeichniseintrags) setzt das Finden des jeweiligen Eintrags voraus, so dass hier ohne Index ein Zeitaufwand von $O(n)$ bzw. $O(\log n)$ entsteht:

	Ohne Index		Master/Slave-Index	
	FAT	ext2	FAT	ext2
Suche	$O(n^2)$	$O(n \log n)$	$O(n)$	$O(n)$
Neue Datei	$O(n)$	$O(\log n)$	$O(n)+O(n)$	$O(\log n)+O(1)$
Änderung	$O(n)$	$O(\log n)$	$O(n)+O(n)$	$O(\log n)+O(n)$
Löschen	$O(n)$	$O(\log n)$	$O(n)+O(n)$	$O(\log n)+O(n)$

Grün hervorgehoben: Verbesserungen bei häufigen Operationen
Rot hervorgehoben: Verschlechterungen bei seltenen Operationen

Abb. 5.2-2: Operationen im Vergleich

Die Verschlechterung beim Ändern und Löschen von Dateien innerhalb moderner Dateisysteme wird akzeptiert, um im Gegenzug eine schnelle Dateisuche zu ermöglichen. Um diesen Nachteil auszugleichen, bietet der Master/Slave-Index eine effiziente Unterstützung für Massen-Operationen wie »batch rename« und »batch delete« (→ *Kap. 5.1.6*), so dass z.B. das Löschen vieler Dateien nur unwesentlich mehr Zeit benötigt als das Löschen einer einzigen Datei.

5.2.1 Optimierung

Das Einführen von Slave-Indexen für jedes Dateiformat lässt das Verfahren ineffizient erscheinen, denn eine Library mit sehr viele Dateitypen benötigt auch sehr viele Slave-Indexe. Bei pragmatischer Betrachtung stimmt dies jedoch nicht, da jede Datei in höchstens einem Slave-Index vorkommen kann. Dadurch können keine Kollisionen entstehen, wenn ein Slave-Index für mehr als ein Dateiformat benutzt wird.

Offenbar ist das Verwenden eines Slave-Indexes für mehrere Dateiformate genau dann kein Problem, wenn die Metadaten-Tupel der jeweiligen Dateitypen dieselbe Größe haben. Ist dies nicht der Fall, können kürzere Tupel durch Opfern von Speicherplatz in der Größe angeglichen werden. Abhängig vom Dateityp werden die Binärdaten des Slave-Index als Tupel mit den jeweiligen Metadaten interpretiert.

Um den zusätzlichen Speicherbedarf bei einer solchen Implementierung zu minimieren, sollten bei Anwendung dieser Technik die Slave-Indexe nicht mehr nach Dateiformat, sondern exponentiell nach Größenklassen eingerichtet werden, also z.B. für Dateitypen mit bis zu 128 Byte an Metadaten, 256 Byte, 512 Byte und so weiter. In der kleinsten Klasse werden somit höchstens 127 Byte verschwendet, in der nächsten Klasse ebenfalls (da ja mindestens 129 Byte belegt sind, damit ein Tupel dieser Klasse zugeordnet wird), anschließend 255 Byte, 511 Byte und so fort. Der maximal verschwendete Speicherplatz entspricht bei exponentieller Klasseneinteilung ab der zweiten Klasse also immer $(0{,}5 * \mathit{Größe}) - 1$, was relativ zur Klassengröße konstant ist.

5.2.2 Verifizierung

Der Merge-Join während der Bearbeitung eines Master/Slave-Indexes wird über den Primärschlüssel F gebildet. Da der Schlüssel zu den Metadaten einer Datei gehört und auch außerhalb des Indexes von Bedeutung ist, muss er zumindest im Master-Index gespeichert werden.

In den Slave-Indexen ist das wiederholte Speichern des Filekeys jedoch nicht erforderlich, da der Merge-Join implizit über die Reihenfolge der Tupel in ihren jeweiligen Heap-Dateien berechnet wird. Um Speicherplatz zu sparen, kann daher durch Anpassung von (5.6) auf das Wiederholen des Schlüsselattributs in den Slave-Indexen verzichtet werden.

Wird von dieser Optimierung kein Gebrauch gemacht, so kann während des Merge-Joins überprüft werden, ob sich an der aktuellen Stelle eines Slave-Indexes auch tatsächlich das angeforderte Tupel befindet. Ist dies nicht der Fall, so ist der Index inkonsistent und muss neu aufgebaut werden.

6 Referenz-Library

Dieses Kapitel stellt die Architektur (→ *Kap. 6.1*) und Implementierung (→ *Kap. 6.2*) einer Referenz-Library vor. Auf diesem System basierend kann schließlich die Interaktion mit Dateien (→ *Kap. 7*) verbessert werden: insbesondere wird hier der vollständige Zugriff auf beliebige Metadaten realisiert, so dass eine starre Verzeichnisstruktur obsolet wird. Zur Sicherstellung einer hohen Leistung wird zur Indexierung ein Master/Slave-Index (→ *Kap. 5*) eingesetzt und erprobt (→ *Kap. 6.3*).

Im Rahmen dieser Arbeit soll auf bestehenden Dateisystemen aufgebaut werden, indem diese um zusätzliche Funktionen erweitert werden. Dabei soll die Architektur skalierbar sein, also auf nahezu beliebige Dateisysteme aufgesetzt werden können. Konkrete, von diesem Modell abgeleitete Libraries können so von Flash-Speicherkarten, die üblicherweise mit dem FAT-Dateisystem formatiert sind, bis hin zu Highend-Systemen eingesetzt werden. Die Library wird dadurch zum integralen Bestandteil des Betriebssystems, so dass Applikationen die bereitgestellte Funktionalität voraussetzen können:

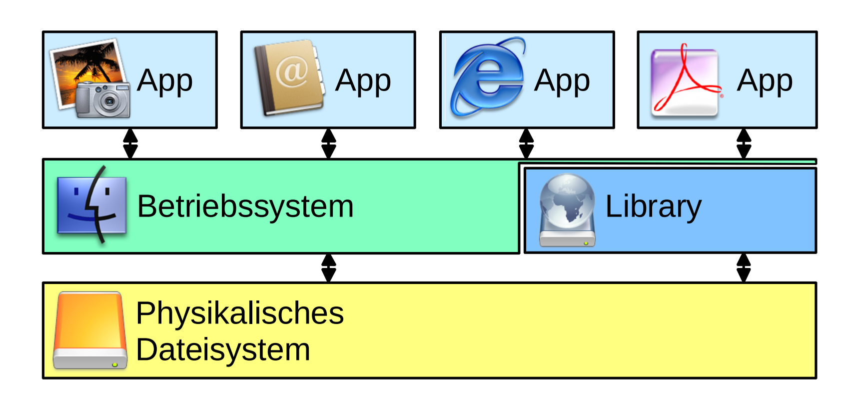


Abb. 6-1: Integration ins Betriebssystem

6.1 Architektur

In diesem Abschnitt wird zunächst die Architektur der Referenz-Library eingeführt, bevor in → *Kap. 6.2* konkrete Details zur Implementierung präsentiert werden.

Die Referenz-Library setzt sich aus mehreren *Domains* (→ *Kap. 6.1.1*) zusammen, die entweder auf einem physikalischen Dateisystem aufbauen oder als »virtuelle Domain« (in → *Abb. 6.1-1* ganz rechts) andere Informationen in die Library integrieren. Applikationen (→ *Kap. 6.1.2*) greifen nicht mehr direkt auf das physikalische Dateisystem zu, sondern werden durch die Library in ein objektorientiertes Gesamtkonzept eingebunden **[Kol08b]**. In einer Registry (→ *Kap. 6.1.3*) werden alle aktiven Komponenten verwaltet:

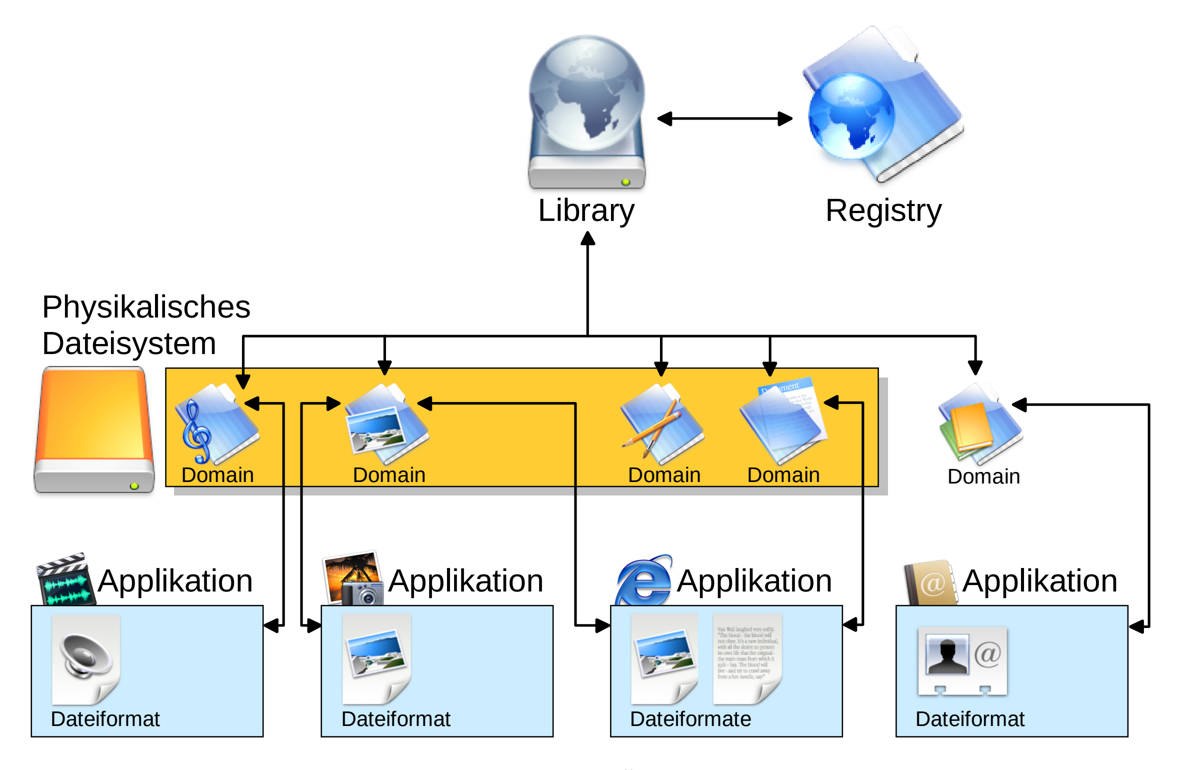


Abb. 6.1-1: Architektur-Übersicht **[Kol08b]**

6.1.1 Domains

Eine Domain **[OSC06]** wird hier als weitgehend unabhängiger Speicherort für Datenobjekte definiert. Die Library wird durch Domains partitioniert: jede Datei ist also in genau einer Domain gespeichert. Der einer Domain zugeordnete Programmcode (→ *Kap. 6.1.2.2*) realisiert den Dateizugriff für Applikationen, indem das physikalische Dateisystem virtualisiert und für das Datenmodell der Library aufbereitet wird **[Kol08b]**.

Viele Dateiformate, etwa für Adressen **[RFC2425]** oder Termine **[RFC2445]**, bestehen nur aus wenigen Attributen und weisen oft keinen Dateikörper auf. Datenobjekte dieser Formate sind kleine Tupel uniformer Größe. Die Referenz-Library sieht für derartige Dateien Domains vor, die alle Tupel eines Typs in einer einzigen Heap-Datei im physikalischen Dateisystem zusammenfassen (je nach Dateiformat beispielsweise ADRESSEN.DAT oder TERMINE.DAT). Der Zugriff auf einzelne Datenobjekte wird vom zugehörigen Programmcode (→ *Kap. 6.1.2.2*) transparent auf die jeweilige Heap-Datei abgebildet **[Kol08b]**. Auf diese Weise wird kein Speicherplatz in Dateisystemen verschwendet, die zur Verbesserung der Schreib- und Leseperformanz große Zuordnungseinheiten einsetzen sollten **[Kol08a]**.

Datenobjekte mit Dateikörper werden in anderen Domains direkt auf Dateien des physikalischen Dateisystems abgebildet. Attribute, die weder das physikalische Dateisystem noch das jeweilige Dateiformat im Dateikörper als Metadaten deklarieren, werden in Headern gespeichert, die jeder Datei

vorangestellt werden. Dies geschieht transparent, da für Applikationen durch Addition der Headergröße zum Dateizeiger nur der ursprüngliche Dateikörper sichtbar ist **[Kol08b]**.

Ein weiterer Grund für die Einführung von Domains sind Benutzerkonten, die bei Multiuser-Betriebssystemen verwendet werden. Zu keinem Zeitpunkt sind hier alle Dateien verfügbar: lediglich die Dateien des jeweils angemeldeten Benutzers werden gemountet, außerdem sind bestimmte Systemdateien (etwa Schriftarten) freigegeben. Daher verfügt jeder Benutzer der Referenz-Implementierung über eigene Domains, die zusammen etwa dem Heimverzeichnis in traditionellen Systemen entsprechen. Zusätzlich werden Domains für Systemdateien angelegt, die für alle Benutzerkonten verfügbar sind.

6.1.2 Applikationen

Das Referenz-Modell muss keine Rücksicht auf bestehende Applikationen und die durch den Verlust der Kompatibilität entstehenden Probleme nehmen. Daher wird ein ganzheitlicher, objektorientierter Ansatz eingeführt, bei dem Applikationen durch Vererbung zum Teil der Library werden:

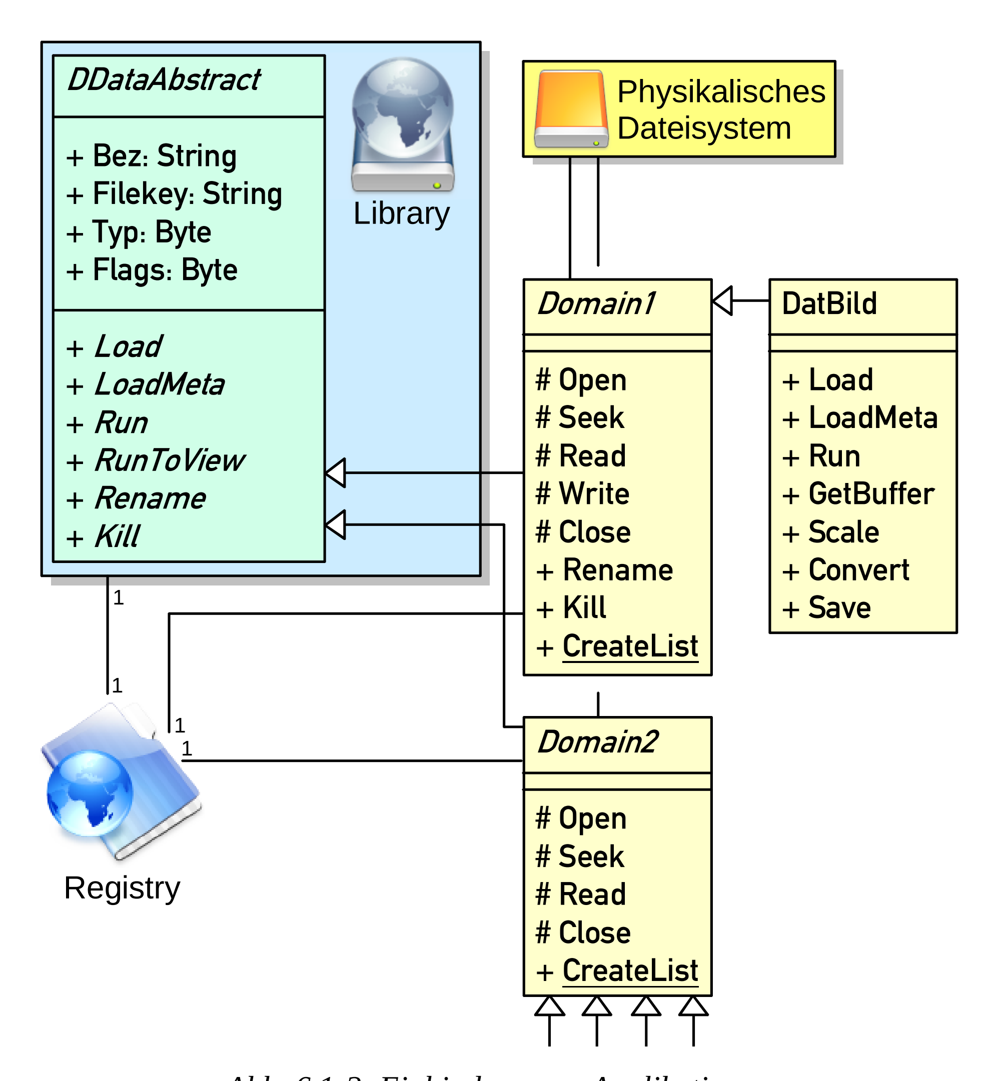


Abb. 6.1-2: Einbindung von Applikationen

6.1.2.1 Datei-Klasse

Oberste Klasse der in → *Abb. 6.1-2* dargestellten Vererbungshierarchie ist die abstrakte Klasse **DDataAbstract**, die die grundlegenden Eigenschaften und Operationen aller Dateiformate modelliert. Neben den obligatorischen Attributen wie dem Primärschlüssel *F* (**Filekey**, → *Kap. 4.1*), dem Typbezeichner *T* (**Typ**, → *Kap. 4.1*), einem Dateinamen **Bez** (für »Bezeichnung«) und diversen Steuerbits (**Flags**) definiert diese Klasse eine Reihe von abstrakten Methoden. Durch Aufruf dieser Methoden können instanziierte Datenobjekte von außen manipuliert, etwa umbenannt, werden. Die konkrete Realisierung, insbesondere die Funktionsweise und das physikalische Format der beteiligten Domains, bleibt dem Aufrufer dabei verborgen.

6.1.2.2 Domain-Klassen

Von der abstrakten Klasse **DDataAbstract** wird auf der nächsten Ebene der Programmcode aller Domains abgeleitet. Domain-Klassen implementieren Methoden für den Dateizugriff, etwa zum Öffnen und Schließen von Dateien (**Open** und **Close**), zum Verschieben des Dateizeigers (**Seek**) sowie zum Lesen und Schreiben von Daten an der aktuellen Position des Dateizeigers (**Read** und **Write**). Diese Operationen werden in geeigneter Weise (→ *Kap. 6.1.1*) auf das physikalische Dateisystem abgebildet und sind nur in der Domain-Klasse selbst sowie allen abgeleiteten Klassen sichtbar (**protected**).

Darüber hinaus stellen Domain-Klassen Methoden zum Umbenennen (**Rename**) und Löschen (**Kill**) von Dateien bereit, sowie eine Methode **CreateList** zur Dateisuche. Diese Methode bekommt eine Suchanfrage (→ *Kap. 6.2.2*) übergeben und liefert eine Liste aller enthaltenen Dateien zurück, die den Bedingungen der Suchanfrage genügen. Die konkrete Implementierung kann variieren, beispielsweise durch die Verwendung eines Indexes (→ *Kap. 5*). Ein solcher Index kann ohne Beteiligung von Applikationen verwaltet werden, da der Zugriff auf Dateien ausschließlich über die von der Domain bereitgestellten Methoden durchgeführt wird. Auf diese Weise kann etwa bei jedem Aufruf von **Close** nach vorhergehenden Schreiboperationen automatisch eine Aktualisierung des Indexes vorgenommen werden.

6.1.2.3 Applikations-Klassen

Formatspezifische Methoden, etwa zum Einlesen einer Datei (**Load**) oder zur Extraktion der Metadaten (**LoadMeta**), bleiben in Domain-Klassen weiterhin abstrakt. Sie werden erst in den eigentlichen Applikations-Klassen implementiert. Die Objekte von Applikations-Klassen repräsentieren damit eine konkrete Datei, die nach dem Aufruf von **Load** in aller Regel in den Arbeitsspeicher eingelesen wurde und zur Benutzung bereitsteht. Alternativ kann ein solches Datei-Objekt auch von einem Domain-Objekt erzeugt worden sein, um z.B. bei einer Neuindexierung durch Aufruf der Methode **LoadMeta** nur die Attributwerte einzulesen und danach in einem Index zu speichern.

Für Applikationen, die Dateien aus mehreren Domains verarbeiten sollen, ergibt sich hier bei konsequenter Umsetzung des Modells das Problem der Mehrfachvererbung, die von vielen Programmiersprachen nicht unterstützt wird. Da aber zu jedem Zeitpunkt höchstens eine Datei pro Applikations-Objekt aktiv ist, ist auch nur der Zugriff auf die Methoden eines einzigen Domain-Objekts erforderlich. Daher kann die Mehrfachvererbung durch dynamisches Linken realisiert werden.

6.1.3 Registry

Die eigentliche Referenz-Implementierung verwaltet eine begrenzte Anzahl vorher definierter Dateiformate (→ *Kap. 6.2.1*); sie ist also statisch. Im Gegensatz dazu muss für praktische Anwendungen die Erweiterbarkeit sichergestellt werden: es müssen also nachträglich neue Datentypen, Domains und Applikationen hinzugefügt werden können. Aus diesem Grund ist in der Architektur eine Registry vorgesehen, die analog zur Registry von Microsoft OLE (→ *Kap. 2.4.2*) funktioniert. Neben einer Liste aller Dateiformate enthält die Registry für jeden Dateityp eine Liste mit Domains, die das jeweilige Dateiformat enthalten können. Darüber hinaus werden für alle Dateiformate die Applikationen, die das jeweilige Format öffnen können, registriert.

6.2 Implementierung

In diesem Abschnitt werden zwei wichtige Teilbereiche der Referenz-Implementierung präsentiert: der Namensraum (→ *Kap. 6.2.1*) und die Suche nach Dateien mit bestimmten Eigenschaften (→ *Kap. 6.2.2*).

6.2.1 Namensraum

Das Datenmodell einer Library (→ *Kap. 4.1*) schreibt für jedes Objekt-Schema R die Attribute F und T vor (4.1), die den Typ eines Datenobjekts global eindeutig codieren (T, 4.6) bzw. für einen gegebenen Typ die Funktion eines Primärschlüssels übernehmen (F, 4.5). Damit bilden die Attributwerte $\langle f, t \rangle$ aller Datenobjekte einen globalen Superschlüssel (4.12). In (4.7) wird gefordert, dass $dom(F)$ und $dom(T)$ für alle Objekt-Schemata einer Library L identisch und somit vergleichbar sein müssen.

In der Praxis kann der Wertebereich $dom(T)$ beispielsweise ein dreibuchstabiger Code sein, wodurch die üblichen Namensendungen weiter benutzt werden können. Ebenso ist auch ein FourCC wie bei AVI-Dateien **[Kol03]**, eine GUID **[RFC4122]** oder ein MIME-Typ **[RFC2045]** denkbar **[Kol08b]**. In der Referenz-Library wurde für $dom(T)$ der Datentyp Byte gewählt, so dass jedes Dateiformat als 8 Bit große Zahl codiert wird. Dadurch können für Operationen mit Dateitypen die Mengenfunktionen von Borland Pascal ausgenutzt werden. Unter anderem sind in der Referenz-Library gegenwärtig folgende Dateiformate definiert worden (das Präfix **dt** steht für »data type«):

```
const
  dtNone=0;             {Ungültig bzw. nicht initalisiertes Datenobjekt}
  dtFolder=1;           {Ordner, interne Verwendung im Datei-Manager (→ Kap. 7.4)}
  dtBild256=6;          {Unkomprimiertes Bild mit 256 Farben}
  dtAdresse=9;          {Adresse}
  dtTermin=11;          {Termin}
  dtText=13;            {Textdatei}
  dtHelp=14;            {Hilfetext}
  dtTexture=15;         {3D-Textur}
  dtAudioCD=16;         {Trackliste einer Audio-CD}
  dtHTML=18;            {HTML-Text}
  dtMail=19;            {EMail}
  dtArchive=22;         {Archivdatei}
  dtMP3=23;             {MP3-Klang}
  dtVideoMPEG=25;       {MPEG-Video}
  dtBildTrue=26;        {Unkomprimiertes Echtfarben-Bild}
  dtSMS=33;             {SMS}
  dtDigiFoto=34;        {JPEG-Bild}
  dtVideoAVI=35;        {AVI-Video}
  dtTrueType=36;        {Skalierbare Schriftart}
  dtQuickTime=39;       {Quicktime-Video}
  dtSearch=42;          {Abgespeicherter Suchfilter (→ Kap. 7.3)}
  LCARSFiletypes=44;    {Bislang höchste vergebene Nummer}
```

Der Wertebereich *dom*(*F*) der Referenz-Implementierung besteht aus genau 8 Zeichen und entspricht damit dem Datentyp **String[8]**. Da das Attribut *F* nur für ein einziges Dateiformat eine Schlüsselfunktion übernehmen muss, können Applikations- bzw. Domain-Klassen die Attributwerte *f* unabhängig vergeben. In der Referenz-Implementierung wird für in Heap-Dateien zusammengefasste Datenobjekte das physikalische Offset innerhalb der Heap-Datei in hexadezimaler Schreibweise verwendet, andernfalls ist der Schlüssel eine zufällig gewählte Zeichenkette. Folgende Abbildung zeigt eine Liste verschiedener Dateien, in der u.a. die Attribute **Bezeichnung** und **Filekey** aufgeführt sind (Dateien mit Hashcodes als Primärschlüssel sind rot markiert):

Dateien (Benutzerdateien) 02:57 ?

Schließen
Filter
Ansicht
Speicher
Drucken
Öffnen
Bearbeiten
Klonen
Passwort
Umbenennen
Löschen
Export
Ins Archiv
Bewerten
Ort
Brennen
Diashow

Dateiname	Dateizeit	ID
Felizitas 23	Unverändert	00006270
Felizitas 24	08.03.2006 12:40	00007CB0
Felizitas 25	Unverändert	00007B98
Felizitas 26	Unverändert	00008C00
Felizitas 27	Unverändert	000087A0
Felizitas 28	Unverändert	00008688
Felizitas 29	Unverändert	00009060
Felizitas Ottemeier	07.01.1980 00:00	00000D2A
Ferengi	27.09.2003 15:05	JE3GIMYY
Flug DUS-FRA-HEL	31.07.2007 00:00	00000056
Flug HEL-FRA-DUS	06.08.2007 00:00	0000002B
Föderation	27.09.2003 15:05	S1RVOBD(
IBM UltiMotion - Laser	11.10.2004 05:54	99JL3XFY
Ingo-Olaf Schumacher		00000695
Jindra	18.11.2006 02:20	P-NQ!7X8
Jindra 1	21.03.2003 15:41	00002BC0
Jindra 2	11.10.2003 12:31	00002DF0
Jindra 3	Unverändert	00003250
Jindra 4	07.01.2005 10:27	000050F0
Jindra 5	Unverändert	00004DA8
Jindra 6	Unverändert	00005AC8
Jindra 7	Unverändert	000059B0
Jindra 8	Unverändert	00008570

9.949.245 Byte (800x 300 Punkte, Cinepak, ∎)

*Abb. 6.2-1: Dateiname, Dateizeit und ID (F bzw. **Filekey**)*

6.2.2 Selektion

Die Auswahl von Dateien ist eine der wichtigsten Operationen einer Library: zum einen ist das Auswählen bzw. Auffinden bestimmter Dateien die Motivation für die Entwicklung eines derartigen Informationssystems (→ *Kap. 1*), andererseits stellt die Selektion eine Vorstufe vieler anderer Operationen dar, etwa für das (Massen)löschen (→ *Kap. 5.1.6*) nicht mehr benötigter Dateien.

Die heute wichtigste Abfragesprache für Datenbanksysteme ist SQL. Das bedeutet insbesondere, dass viele Entwickler in der Industrie und im akademischen Umfeld mit dieser Sprache vertraut sind. SQL gestattet durch den SELECT-Befehl die Formulierung sehr komplexer Selektionen, so dass ein Parser diese Ausdrücke in mehrere elementare Suchanfragen aufspalten und die Ergebnisse danach verknüpfen muss, etwa durch Vereinigung der entstandenen Mengen **[Saa05]**.

Im Interesse einer möglichst einfachen und effizienten Referenz-Implementierung wird hier jedoch auf SQL als Abfragesprache verzichtet. Zur Selektion von Dateien wird auf den Attributen der Library basierend eine Datenstruktur definiert, die die Parameter einer Suchanfrage speichert und »*Filter*« genannt wird:

```
type
 FilterType=record
 {1. Dateitypen}
   Dateitypen: Filetypes;
 {2. Gemeinsame Attribute}
   Volltextsuche: String[31];          {Für Volltextsuche}
   NameParam: String[31];
   NameSuchmuster: Byte;               {1: enthält, 2:ist, 3:fängt an mit, 4:hört auf mit}
   DatumParam: String[10];
   DatumSuchmuster: Byte;              {1: neuer/gleich, 2:ist, 3:älter/gleich, 4:Jahrestag}
   FlagsParam: Byte;                   {1: Neu, 2: System}
   FlagsSuchmuster: Byte;              {1: ist; 2: ist nicht}
 {3. Typabhängige Metadaten}
   Absender,Empfaenger: String[31];    {Für SMS und EMail}
   Thema: String[31];                  {EMail, AVI}
   KuenstlerParam,Titel: String[31];   {MP3, AVI}
   Erscheinungsjahr,Album: String[31]; {MP3}
   EquipmentFirmware: String[31];      {JPEG, AVI}
   AufnahmeParam: String[10];          {JPEG, AVI}
   AufnahmeSuchmuster: Byte;           {1: neuer/gleich, 2:ist, 3:älter/gleich}
   BreiteParam: String[4];             {Bilder, Videos, Animationen}
   HoeheParam: String[4];              {Bilder, Videos, Animationen}
   KuenstlerSuchmuster: Byte;          {1: enthält; 2: ist}
   ...
end;
```

Damit eine Datei ins Suchergebnis aufgenommen wird, müssen alle aktiven Bedingungen der Filterstruktur erfüllt sein, es handelt sich also um eine implizite AND-Verknüpfung. In Sektion 1 wird zunächst die Menge der erlaubten Dateitypen (→ *Kap. 6.2.1*) definiert. Daran schließen sich zwei Sektionen an, die sich auf die weiteren Attribute der Dateien beziehen.

Im zweiten Abschnitt ist für jedes gemeinsame (also in allen Objekt-Schemata enthaltene) Attribut eine Variable vorhanden, zusammen mit einem Byte, das das Suchmuster definiert. Enthält beispielsweise **NameParam** eine Zeichenkette, so legt **NameSuchmuster** fest, ob der Dateiname **NameParam** an beliebiger Stelle enthalten sein soll (**1**), identisch mit **NameParam** (**2**) sein bzw. damit beginnen (**3**) oder enden (**4**) soll. Analog dazu werden für alle anderen Attribute Bedingungen definiert. Zusätzlich enthält die Sektion einen Parameter für die Volltextsuche, der sich auf alle Attribute und bei Textdateien sogar auf den Dateiinhalt bezieht. Der dritte Abschnitt der Filterstruktur enthält Parameter, um Bedingungen für alle typabhängigen Attribute, wie z.B. **Erscheinungsjahr** oder **Titel**, zu definieren.

Zusammenfassend besteht ein Filter also aus einer Menge von Dateitypen und Bedingungen, die für die Aufnahme einer Datei ins Suchergebnis ausnahmslos erfüllt sein müssen. Damit bietet ein Suchfilter weniger Möglichkeiten als SQL, ist aber besonders leicht zu interpretieren und noch mächtig genug für die Formulierung der wichtigsten in der Praxis erforderlichen Suchanfragen. Darüber hinaus gestattet der Eintrag **Volltextsuche** die Angabe eines Suchbegriffs, nach dem in allen Attributwerten und (abhängig vom Dateiformat) sogar im Dateikörper selbst gesucht wird. Ein derartiger Suchfilter ist mit SQL gar nicht bzw. nur umständlich oder mittels proprietärer Erweiterungen zu formulieren.

Die Manipulation des Datenbestandes wird im Referenz-Modell weder über einen Filter noch mittels einer Befehlssprache durchgeführt, sondern durch den Aufruf von Methoden, die Domains bzw. Applikationen bereitstellen (→ *Kap. 6.1.2*).

6.2.2.1 Query containment

Vor der eigentlichen Durchführung einer Dateisuche kann die Filterstruktur optimiert werden, indem Dateiformate aus der Menge **Dateitypen** entfernt werden, die ohnehin keine Suchergebnisse liefern. Diese Optimierung wird als »query containment« bezeichnet **[Mil00]**, also als Eingrenzen der Suche. Folgender Filter verdeutlicht das Prinzip:

```
ResetFilter;                    {Setzt den gesamten Filter zurück}
  with Filter do begin
  Dateitypen:=[dtMP3,dtVideoAVI,dtQuicktime];
  BreiteParam:='640';           {Breite größer oder gleich 640 Pixel}
  HoeheParam:='480';            {Höhe größer oder gleich 480 Pixel}
  KuenstlerParam:='Anastacia';
  KuenstlerSuchmuster:=2;       {ist}
  end;
Retrieve;                       {Startet die Suche}
```

Es ist offensichtlich, dass diese Suche niemals MP3s liefern wird, da dieses Format als reine Audio-Datei nicht über eine Breite oder Höhe verfügt (→ *Kap. 3.3.4*). Daher muss für jede im Filter geforderte Bedingung die Menge **Dateitypen** mit der Menge der Formate geschnitten werden, die das entsprechende Attribut überhaupt besitzen – da **dtMP3** keine Breite oder Höhe besitzt, würde dieses

Element also aus **Dateitypen** entfernt. Optimierte Filter betreffen möglicherweise nicht alle Domains, so dass sie von einem Master/Slave-Index schneller bearbeitet werden können (→ *Kap. 5.1.2*). In der Referenz-Implementierung wird daher jede Suchanfrage vor ihrer Abarbeitung wie folgt eingegrenzt:

```
Suchtypen=Filter.Dateitypen;          {Kopie, um Struktur des Aufrufers nicht zu verändern}
{Suchtypen bereinigen: Typen entfernen, denen geforderte Metadaten fehlen}
 with Filter do begin
  if (Absender<>'') or (Empfaenger<>'') then Suchtypen:=Suchtypen*[dtSMS,dtMail];
  if Thema<>'' then Suchtypen:=Suchtypen*[dtMail,dtVideoAVI];
  if (Kuenstler<>'') or (Titel<>'') then Suchtypen:=(Suchtypen*[dtMP3,dtVideoAVI]);
  if (Erscheinungsjahr<>'') or (Album<>'') then Suchtypen:=Suchtypen*[dtMP3];
  if EquipmentFirmware<>'' then Suchtypen:=Suchtypen*[dtDigiFoto,dtVideoAVI];
  if AufnahmeSuchmuster>1 then Suchtypen:=Suchtypen*[dtDigiFoto,dtVideoAVI];
  if (BreiteParam<>'') or (HoeheParam<>'') then
   Suchtypen:=Suchtypen*[dtFlic,dtXAnim,dtVideoMPEG,dtVideoAVI,dtQuickTime,
    dtBildTrue,dtBild256,dtBildAnsi,dtDigiFoto,dtTexture,dtVideoSMJPEG];
  ...
 end;
```

6.3 Performanz

Im Rahmen der Referenz-Implementierung wurde der Master/Slave-Index aus → *Kap. 5* einem Performanztest unterzogen. Die Messergebnisse unterstreichen die theoretische Leistungsfähigkeit der Indexstruktur im Zusammenspiel mit der Referenz-Implementierung, deren Praxistauglichkeit dadurch nachgewiesen wird.

6.3.1 Testumgebung

Zum Einsatz kamen wieder die Testdaten und das Testsystem aus Abschnitt → *Kap. 3.4*. Das Testsystem war mit einer CPU vom Typ AMD Athlon 64 X2 3800 dual core (2000 MHz), 2 GB DDR2 dual channel RAM (200 MHz Bustakt) und einer 320 GB SATA-Festplatte ausgestattet, die als NTFS-Dateisystem formatiert wurde. Zeitmessungen wurden mit dem **RDTSC**-Befehl **[She96]** durchgeführt.

6.3.2 GREP

Für einen ersten Geschwindigkeitstest wurde der Befehl **GREP** benutzt, der Dateien nach bestimmten Zeichenketten durchsucht. **GREP** verfügt über keine Informationen zu den einzelnen Dateiformaten, sondern öffnet jede Datei und durchsucht den gesamten Dateikörper. Das impliziert eine enorme Zeitverschwendung bei großen Multimedia-Dateien, die Metadaten nur in kleinen Bereichen am Anfang oder Ende der Datei enthalten **[EXI07] [Nil05]**.

Bei 1442 Dateien benötigte GREP 955,3 Sekunden (also mehr als 15 Minuten) für Testfall 5 (alle MP3-Dateien von einer bestimmten Künstlerin). Da solche Laufzeiten in der Praxis unbrauchbar sind, wurde auf weitere Tests mit größerem Datenbestand verzichtet **[Kol08c]**.

6.3.3 Ohne Index

Im nächsten Test wurden alle fünf Testfälle mit der Referenz-Library bearbeitet, allerdings ohne einen Index. Die Laufzeit ist gegenüber GREP (→ *Kap. 6.3.2*) deutlich geringer, da die Referenz-Library die jeweiligen Dateiformate korrekt parsen kann und nur diejenigen Bereiche einliest, die tatsächlich relevante Metadaten enthalten. Auf diese Weise ergibt sich eine etwa hundertfache Geschwindigkeitssteigerung gegenüber GREP:

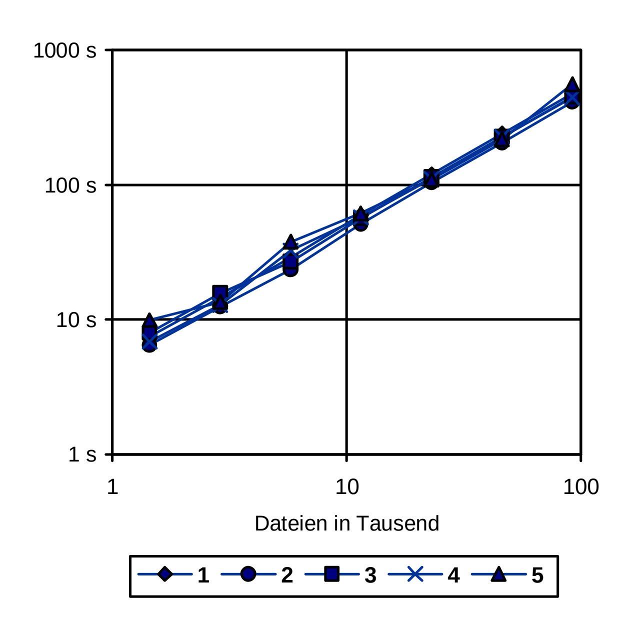


Abb. 6.3-3: Laufzeiten ohne Indexierung **[Kol08c]**

Die Laufzeit ist im Wesentlichen von der Anzahl der Dateien abhängig, so dass die Testfälle linear skalieren. Da die Verzeichniseinträge von NTFS in einem B-Baum gespeichert werden, ist theoretisch eine Laufzeit von $O(n \log n)$ zu erwarten. Die Verzeichniseinträge werden aufgrund des häufigen Zugriffs jedoch im RAM gecacht, so dass hier kaum Blöcke von der Festplatte gelesen werden.

6.3.4 Master/Slave-Index

Im nächsten Schritt wurde die Referenz-Library um einen Master/Slave-Index erweitert. Wiederum wurden alle fünf Suchanfragen auf verschieden großen Datenbeständen bearbeitet. Gegenüber einer Implementierung ohne Indexierung ist eine deutliche Steigerung der Geschwindigkeit messbar, wiederum etwa um Faktor 100:

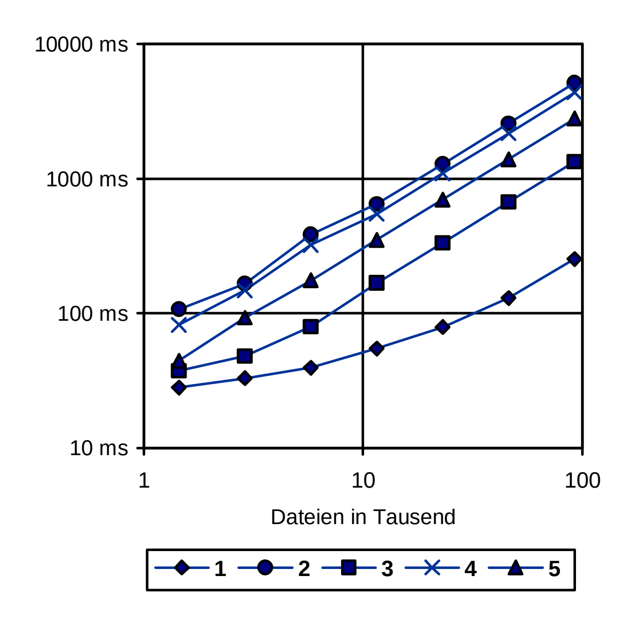


Abb. 6.3-4: Laufzeiten mit Master/Slave-Index **[Kol08c]**

Das Bearbeiten des ersten Testfalls hat einen besonders geringen Zeitaufwand, da das zu prüfende Attribut im Master-Index gespeichert wird, und daher keine Slave-Indexe bearbeitet werden müssen.

6.3.5 Komprimierter Master/Slave-Index

Da die Performanz des Master/Slave-Index vor allem von der Geschwindigkeit des Datenträgerzugriffs abhängt, kann die Leistung durch Verringern der zu lesenden Datenmenge gesteigert werden. In diesem Zusammenhang ist die verlustfreie Kompression der Indexdateien besonders interessant. Die Index-Tabellen eines Master/Slave-Index werden ausschließlich sequenziell verarbeitet, so dass der Dateizeiger niemals neu positioniert werden muss. Dadurch können die Datenströme transparent vor der Verarbeitung dekomprimiert und bei Änderungen wieder komprimiert werden.

Eine Komprimierung mit PKZIP (Deflate-Methode, basierend auf dem LZ77-Algorithmus und Huffman-Codierung) erzielt einen Kompressionsfaktor von 16:1, was einer Steigerung der Geschwindigkeit um Faktor 10 entspricht. Der Zeitverbrauch für das Bearbeiten eines komprimierten Master/Slave-Indexes wächst nach wie vor linear mit der Dateigröße, zusätzlich hat das Initalisieren des Dekomprimierers einen konstanten Zeitbedarf:

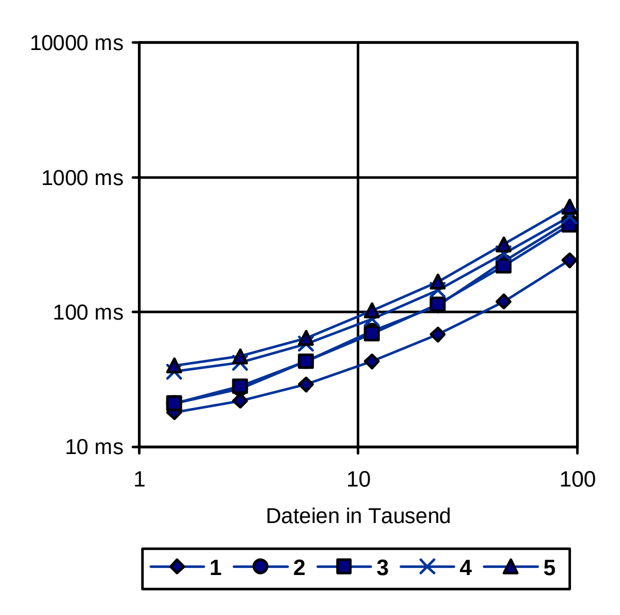


Abb. 6.3-5: Laufzeiten mit komprimiertem Master/Slave-Index **[Kol08c]**

6.4 Vorteile

Die hier vorgestellte Referenz-Library basiert auf dem Datenmodell einer Library (→ *Kap. 4*) bzw. implementiert dieses, wodurch alle in Abschnitt → *Kap. 4.4* genannten Vorteile wie einheitlicher Zugriff auf heterogene Datenquellen und Typsicherheit auch für die Referenz-Implementierung gelten. Die Referenz-Library zeigt darüber hinaus die praktische Funktionsfähigkeit des Datenmodells. Sowohl das Architekturmodell als auch die gewählte Implementierung bieten außerdem noch weitere Vorteile.

6.4.1 Architektur

Die objektorientierte Architektur der Referenz-Library bietet gegenüber anderen Systemen viele Vorteile. Zunächst kann der Zugriff auf Dateiattribute kompakt durch Domain-Klassen realisiert werden, die zur Extraktion von Metadaten den Programmcode der Applikationen wiederverwenden können. Bei einer vollständige Neuindexierung wird für jede Datei die Methode **LoadMeta** der zugehörigen Applikation aufgerufen, die alle Attributwerte der jeweiligen Datei als Datenstruktur zurückliefert. Beim eigentlichen Zugriff auf Dateien wird ein Index eingesetzt.

Darüber hinaus können Applikationen anderen Modulen zusätzliche Methoden zur Manipulation einer geladenen Datei bereitstellen. Das soll am Fallbeispiel einer Applikation erläutert werden, die Flugpläne speichert und alle absolvierten Flüge auf einer Weltkarte darstellen kann. Eine so erzeugte Weltkarte kann als Bilddatei gespeichert werden, um beispielsweise in einem Fotolabor als Poster ausbelichtet zu werden:

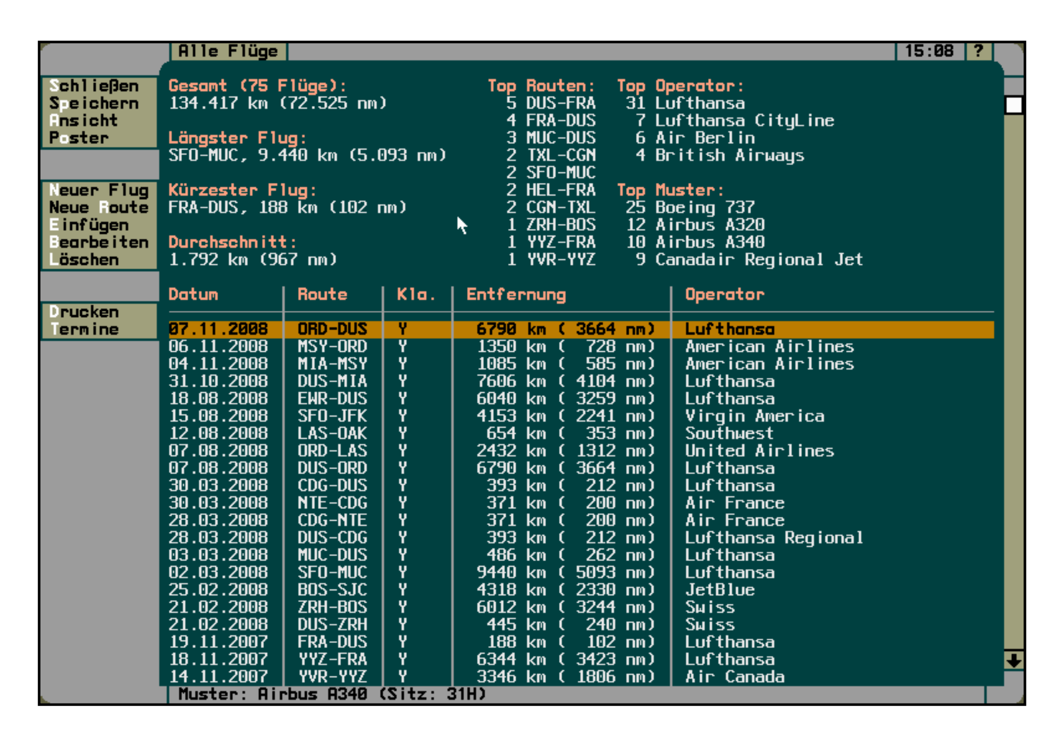


Abb. 6.4-1: *Verwaltung von Flugplänen*

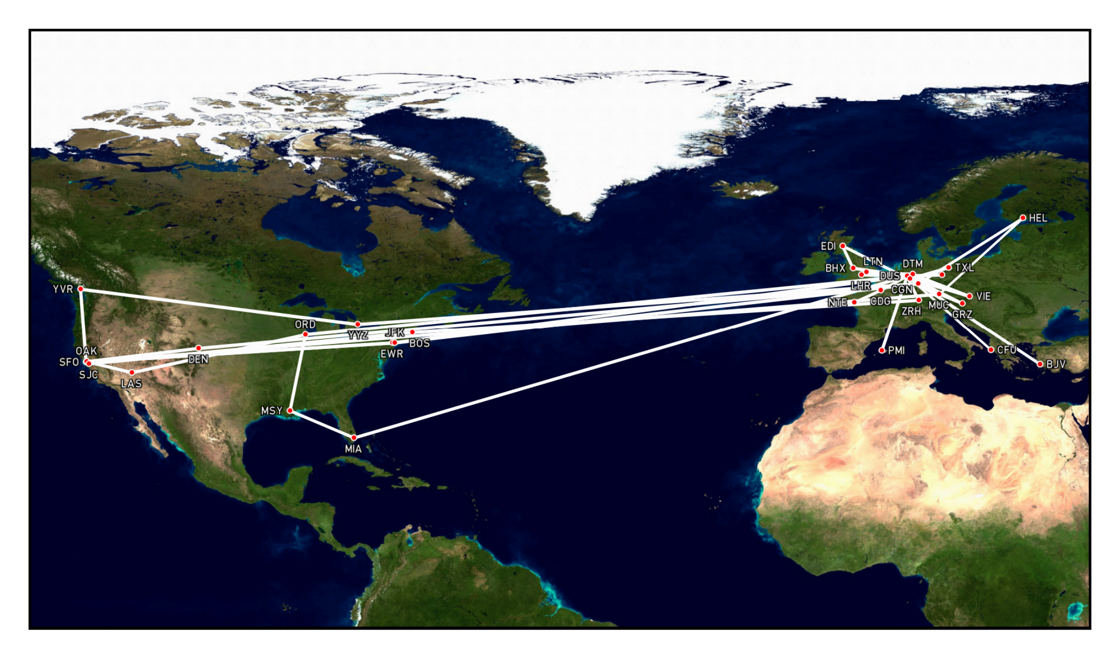


Abb. 6.4-2: *Darstellung eines Flugplans als Poster*

Zur Berechnung einer Bilddatei muss die Flugplan-Applikation zunächst ein Hintergrundbild laden, das in der Referenz-Implementierung als Systemdatei vom Typ **dtDigiFoto** (→ *Kap. 6.2-1*) verfügbar ist. Diesem Dateiformat ist die Applikation **DatBild** zugeordnet. Diese Klasse enthält u.a. die öffentlichen Methoden **Scale**, **Convert**, **Save** und **GetBuffer** zum Skalieren, Konvertieren und Speichern der Datei sowie zum Manipulieren der Bildinformationen.

Bei traditioneller Programmierung müsste die Flugplan-Applikation das Hintergrundbild selbst einlesen, die Flugwege über das Bild kopieren, die entstandene Weltkarte mit eigenem Programmcode auf die vom Benutzer gewünschte Auflösung skalieren und danach in einem bestimmten Format

(z.B. wieder als JPEG-Datei) abspeichern. Unter anderem müsste die Applikation also einen JPEG-Decoder und -Encoder enthalten, darüber hinaus eine Programmroutine zum Skalieren von Bildern.

Durch konsequente Ausnutzung der Referenz-Architektur ist die Flugplan-Applikation tatsächlich wesentlich kompakter. Zunächst wird ein Objekt von **DatBild** erzeugt, in das das Hintergrundbild geladen wird. Die öffentliche Methode **GetBuffer** liefert einen Zeiger auf die Bilddaten zurück, so dass die Flugwege über das Bild kopiert werden können. Danach wird mit **Scale** die Auflösung angepasst und die fertige Bilddatei mit **Save** gespeichert. Die Klasse **DatBild** stellt diese Methoden zur Verfügung, die so nicht noch einmal implementiert werden müssen. Verfügt eine neuere Version von **DatBild** beispielsweise über einen schnelleren JPEG-Decoder, profitieren so auch andere Programme davon.

6.4.2 Implementierung

Die Implementierung der Referenz-Library bietet weitere Vorteile gegenüber anderen Systemen. Der Verzicht auf SQL als Abfragesprache und die Einführung der Filter-Datenstruktur (→ *Kap. 6.2.2*) ermöglicht die Portierung auf mobile Plattformen mit eingeschränkter Rechenleistung bei gleichzeitiger Unterstützung einer Volltextsuche. Durch die gleichzeitige Verwendung eines Master/Slave-Indexes (→ *Kap. 5*) wird eine hohe Leistung bei Suchanfragen sichergestellt (→ *Kap. 6.3.5*).

7 Interaktion

In diesem Kapitel werden die vielfältigen Vorteile für die Organisation und Interaktion von Daten präsentiert, die sich aus dem Einsatz einer Library als Speichersystem ergeben. Als Beispiel wird dabei die Referenz-Library (→ *Kap. 6*) eingesetzt. Die Interaktion mit einer Library lässt sich allgemein auf vier Pfade aufteilen:

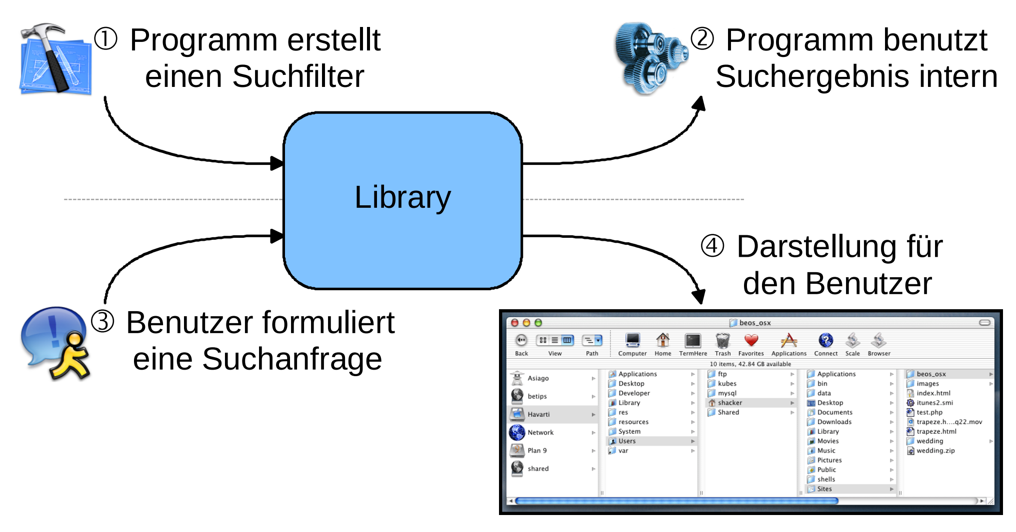


Abb. 7-1: Interaktion mit einer Library

1. Suchanfragen können intern von einem Programm gestellt werden, ohne dass der Benutzer dazu einen Suchfilter (→ *Kap. 6.2.1*) spezifizieren muss. Beispiele hierfür sind eine speziell entwickelte *Shell* (→ *Kap. 7.1*), die u.a. nach neuen Dateien und fälligen Terminen sucht, und Join-Operationen innerhalb von Applikationen (→ *Kap. 7.2*).
2. Manche Programme zeigen das Suchergebnis nicht als Dateiliste an, sondern verarbeiten es intern weiter. Die in → *Kap. 7.1* präsentierte Shell dient auch hier als Beispiel, da die gefundenen Dateien nur gezählt, nicht aber aufgelistet werden werden. Weitere Programme, die von Pfad 2 Gebrauch machen, könnten Hilfsprogramme sein, die in regelmäßigen Abständen alte, bereits gelesene EMails löschen bzw. archivieren, sowie Programme zum Massenlöschen oder -umbenennen von Dateien (→ *Kap. 5.1.6*).
3. Neben den vordefinierten Suchfiltern sollen Benutzer auch selbst Suchabfragen formulieren können, wenn die vordefinierten Suchfilter der Shell nicht ausreichen. In → *Kap. 7.3* wird daher ein Dialogfenster zur Eingabe von Suchfiltern vorgestellt.
4. Ebenso wichtig ist die Darstellung von Dateilisten, etwa eines Suchergebnisses, für den Anwender. Hier ergeben sich durch die einer Library inhärenten Semantik diverse Vorteile und Verbesserungen, unter anderem automatische Ordner (→ *Kap. 7.4*), Anzeigen von Aufgaben (→ *Kap. 7.6*) und verbesserte Webserver (→ *Kap. 7.7*).

Da die in diesem Kapitel vorgestellten Verbesserungen zum Teil völlig unterschiedliche Problemstellungen bearbeiten und daher nicht vergleichbar sind, werden ihre Vorteile nicht am Ende dieses Kapitels zusammengefasst, sondern am Ende der jeweiligen Abschnitte.

7.1 Shell

Der Begriff »Shell« stammt von Unix-Betriebssystemen und bezeichnet die dort verwendeten Kommandointerpreter. Da unter Unix Shells gewöhnliche Programme ohne besondere Rechte sind, entstanden viele gleichberechtigt nebeneinander stehende Shells, die sich teilweise erheblich in Syntax und Funktionsumfang unterscheiden **[Rob99]**. Auch bei ausschließlich grafisch orientierten Betriebssystemen existiert eine Shell. Sie interpretiert dort jedoch keine eingegebenen Kommandos, sondern ist in der Regel das Anwendungsprogramm, das nach dem Booten als erstes gestartet wird. Welches Programm als Shell fungieren soll, lässt sich bei vielen Betriebssystemen einstellen.

In diesem Abschnitt wird eine Shell entworfen, welche die erweiterten Möglichkeiten einer Library unterstützt. Zunächst wird in → *Kap. 7.1.1* der Funktionsumfang moderner Shells vorgestellt, bevor in → *Kap. 7.1.2* Anforderungen an eine Shell auf Library-Basis formuliert werden. In → *Kap. 7.1.3* werden Mechanismen zur Erfüllung dieser Anforderungen eingeführt.

7.1.1 Moderne Shells

Sehr häufig verwendete grafische Shells sind der »Explorer« (→ *Kap. 2.6*), der seit Microsoft Windows 95 Bestandteil des Betriebssystems ist, und der »Finder« als Shell von Mac OS. Für Unix stehen neben den Kommandozeilen-Shells verschiedene grafische Shells als Teil von Desktop-Managern wie KDE **[KDE06]** oder Gnome **[Gno06]** zur Verfügung. Zum Funktionsumfang moderner Shells gehört in der Regel ein Menu zum Starten von Programmen und Funktionen, das neuerdings durch auf dem Desktop platzierte *Widgets* ergänzt wird, sowie ein Datei-Manager.

7.1.1.1 »Start«-Menu und Widgets

Die Shell-Komponente, die zur Auswahl von Befehlen und Programmen dient, ist in den meisten Betriebssystemen als ähnlich aufgebautes Menu angelegt, das in der Regel durch Klick auf eine besondere Schaltfläche (z.B. »Start« in der linken unteren Bildschirmecke) geöffnet wird:

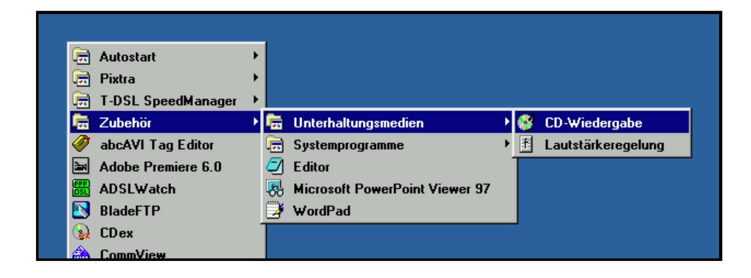


Abb. 7.1-1: »Start«-Menu von Windows 98 (Ausschnitt)

Leider wird die Menuhierarchie bei sehr vielen installierten Programmen schnell unübersichtlich und schwer handhabbar. Zudem ist das Layout eines solchen Menus starr: wie in → *Abb. 7.1-1* zu

sehen ist, steht für jeden Menupunkt nur ein kleines Symbol und eine Zeile Text zur Verfügung. Das Layout ist für alle Optionen identisch, so dass besonders wichtige oder häufig benutzte Menupunkte nicht hervorgehoben werden können.

Diese Unzulänglichkeit und der gleichzeitge Wunsch vieler Benutzer, Informationen oder Alarme wie neu eingetroffene EMails ständig im Blickfeld zu haben, hat zur Beliebtheit von Widgets beigetragen. Dabei handelt es sich um kleine Programme, die ohne die Kontrollelemente eines Fensters direkt auf dem Desktop dargestellt werden und diverse Aufgaben, etwa die Anzeige der Uhrzeit, der CPU-Auslastung oder auch von Aktien-Kursen, übernehmen:



Abb. 7.1-2: Widgets in Windows Vista

7.1.1.2 Datei-Manager

Die zweite wichtige Komponente moderner Shells dient der Organisation des Dateisystems. Der Datei-Manager des Explorers besteht aus einem Fenster mit dem Titel »Arbeitsplatz«, das einen vom Speicherort abhängigen Einstieg ins Dateisystem bietet; dies ist besonders sinnvoll, da der Dateiname mit Pfad den Primärschlüssel eines traditionellen Dateisystems darstellt und somit ein wichtiges Ordnungskriterium ist.

Es sei darauf hingewiesen, dass die in → *Abb. 7.1-3* am linken Fensterrand untergebrachten »Favorite Links« nicht etwa eine typspezifische Suche oder Filterung aktivieren, sondern lediglich die von Windows bei der Installation angelegten Verzeichnisse **Documents** (in der deutschen Version **Eigene Dateien** genannt), **Documents\My Pictures** (**Eigene Dateien\Eigene Bilder**) bzw. **Documents\My Music** (**Eigene Dateien\Eigene Musik**) öffnen, unabhängig von ihrem tatsächlichen Inhalt:

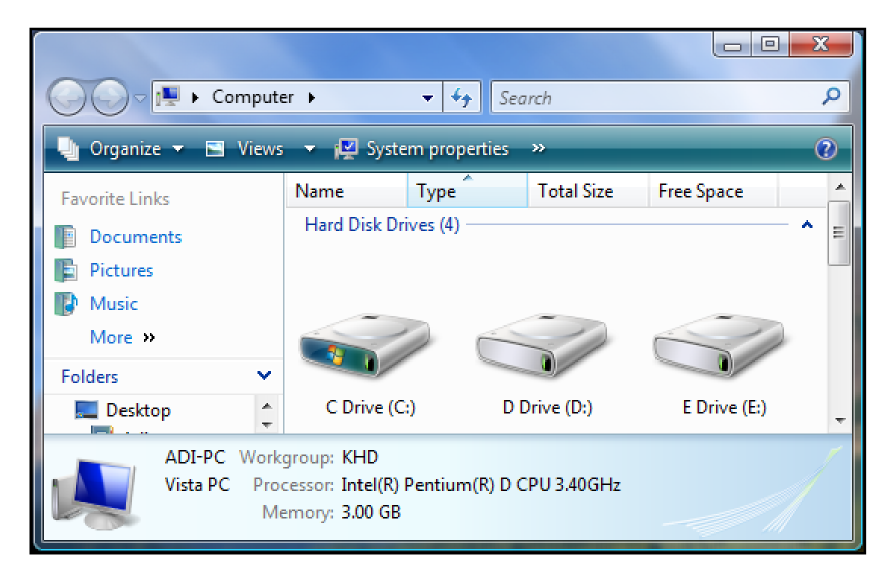


Abb. 7.1-3: »Computer«-Fenster von Windows Vista

7.1.2 Anforderungen an eine Shell

Soll eine Library zur Datenspeicherung benutzt werden, so erscheint die Präsentation von Verzeichnissen als Einstieg ins Dateisystem unzureichend, da eine derartige Hierarchie nicht mehr angeboten wird. Außerdem stehen beliebige Suchfilter bereit, die jedoch nicht genutzt werden. Daher lassen sich aus den oben beschriebenen Eigenschaften und Unzulänglichkeiten aktueller grafischer Shells Anforderungen an verbesserte Shell-Generationen herleiten:

1. Verbesserte Menuführung, um auch viele Befehle und Applikationen übersichtlich und mit Hilfetexten bzw. Hinweisen darzustellen
2. Unterstützung einer Library durch verbesserten Aufruf des Datei-Managers
3. Exponierte Darstellung von besonders wichtigen Menupunkten, Hinweisen und Alarmen, um Widgets zu ersetzen

7.1.3 Hypertext-Menus

Die oben gestellten Anforderungen implizieren eine flexiblere Darstellung für einzelne Befehle, die über das uniforme Layout eines Menus hinausgehen. Komplexere Layouts sind den meisten Anwendern bereits durch WWW-Seiten vertraut.

Die Grundidee zur Verbesserung grafischer Shells besteht deshalb darin, Befehle und Anwendungen als Link in Hypertext-Seiten einzubetten. Da die meisten Betriebssysteme über Webbrowser verfügen und derartige Seiten anzeigen können, erscheint es sinnvoll, die Darstellungsmöglichkeiten von HTML um proprietäre Strukturen zu erweitern, die dann innerhalb für Menuseiten verwendet und vom Browser verstanden werden können.

Angaben zum Verkäufer
Verkäufer: konstantinkoll (430)
Bewertungen: **99,1 % Positiv**
Mitglied: seit 18.07.00 in Deutschland
Angemeldet als privater Verkäufer
- Bewertungskommentare lesen
- Frage an den Verkäufer
- Zu meinen bevorzugten Verkäufern hinzufügen
- **Andere Artikel des Verkäufers**

Sicher kaufen
1. **Sehen Sie sich das Bewertungsprofil des Verkäufers an**
Bewertungspunkte: 430 | 99,1% Positiv
Bewertungskommentare lesen
2. **Informieren Sie sich über den Käuferschutz**
PayPal **Kostenloser Käuferschutz**
Wenn Sie PayPal verwenden, sind Ihre Käufe bei eBay bis zu 500 EUR abgesichert. Mehr zum Thema

Abb. 7.1-4: WWW-Seite mit flexiblem Layout für einzelne Befehle

Die Darstellung von Programmen und Dateien als Link innerhalb einer Menuseite ist einfach zu realisieren. Für lokale Dateien definiert HTML das URL-Schema **[RFC3986]** file:// – so lässt sich eine Applikation durch Verlinkung der ausführbaren Binärdatei einbinden:

```
<A HREF="file://C:\Programme\Picture Publisher\PP50.EXE">Picture Publisher</A> starten
```

Abb. 7.1-5: Darstellung eines Programms als HTML-Link

Für Optionen, die nicht mit einer lokalen Datei verbunden werden können, kann ein proprietäres URL-Schema wie menu:// eingeführt werden, das für den Anwender unsichtbar bleibt und nur vom systemeigenen Browser im »Shell-Modus« verstanden wird.

Um die Übersicht einer Menuseite weiter zu erhöhen, wurde in der Referenz-Shell ein Mechanismus implementiert, der Menupunkte gruppiert und zu Sektionen zusammenfasst. Wird HTML zur Definition des Menus verwendet, können Sektionen (auch außerhalb von Menuseiten in anderen Dokumenten) durch Einführung eines neuen <SECTION>-Tags realisiert werden:

```
<SECTION TITLE="Nachrichten">
   ...
</SECTION>
```

Abb. 7.1-6: Definition von Sektionen mit HTML

In einer Referenz-Implementierung eines Hypertext-Menus werden Sektionen nicht nur als optische Gliederung eingesetzt, sondern alle innerhalb einer Sektion befindlichen Elemente können vom Benutzer »eingeklappt« werden, so dass voraussichtlich länger nicht benötigte Optionen versteckt werden. Der Status jeder Sektion wird persistent gespeichert, so dass die Menusektionen ihren Zustand bis zur nächsten Änderung auch über mehrere Logins hinweg beibehalten:

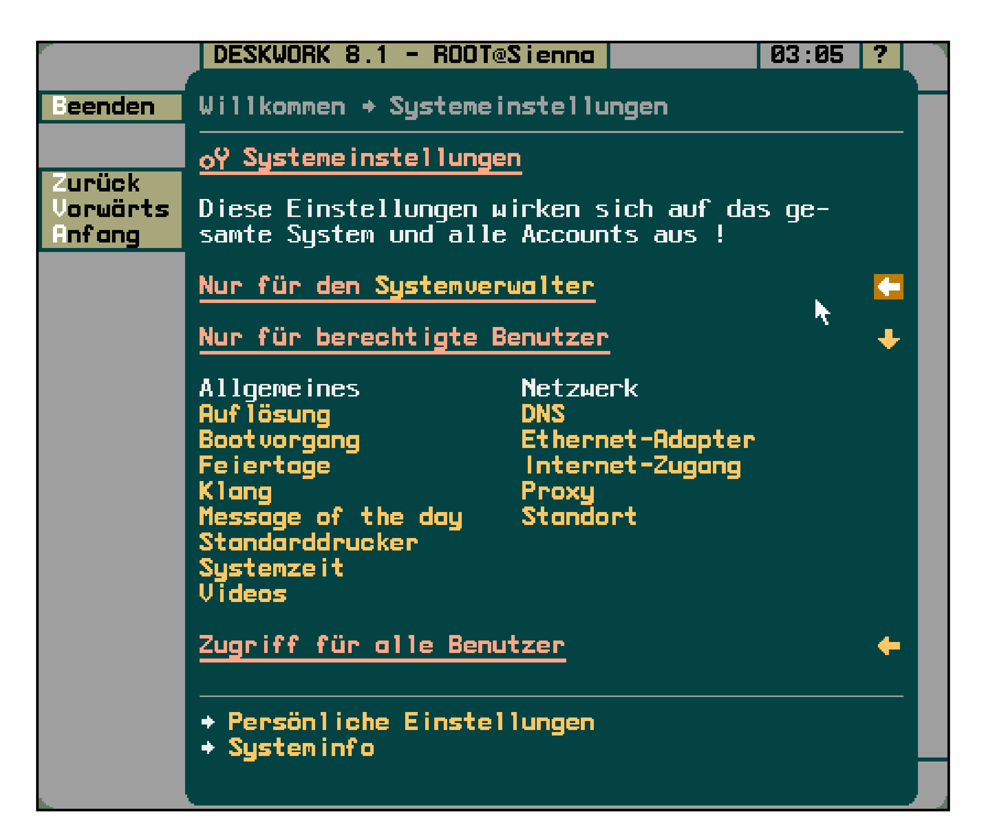


Abb. 7.1-7: Darstellung von Sektionen in einer Menuseite

7.1.3.1 Dateizugriff

Derartige Hypertext-Menus sind vorzüglich geeignet, den erweiterten Funktionsumfang einer Library zugänglich zu machen. Auf der Hauptseite des Menus wurden zwei Sektionen für die Verwaltung von Dateien eingerichtet: »Eigene Dateien« enthält 3 Punkte, die Zugriff auf die Library bieten, während unter »Externe Dateien« Dateisysteme außerhalb der lokalen Library wie eingelegte Medien und angeschlossene externe Datenträger aufgeführt sind:

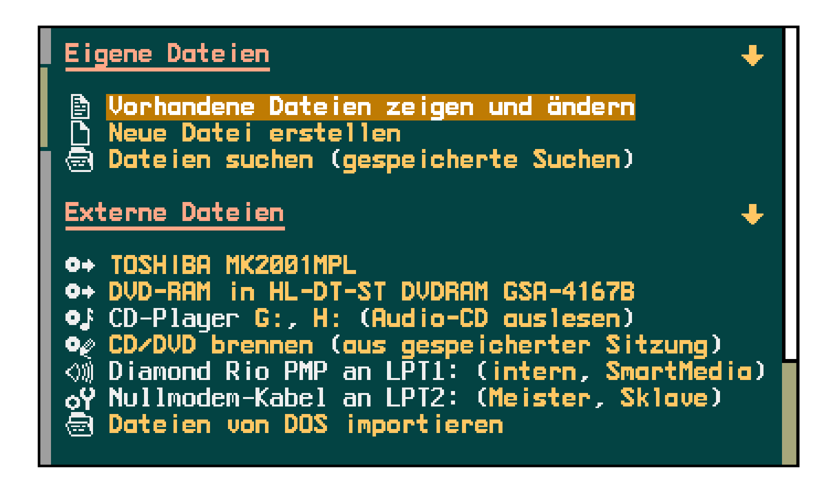


Abb. 7.1-8: »Eigene Dateien«

Die beiden ersten Optionen (→ *Abb. 7.1-8*) verweisen auf Unterseiten, während »Dateien suchen« die Eingabe neuer Suchfilter durch den Benutzer ermöglicht (→ *Kap. 7.3*). »Gespeicherte Suchen« zeigt alle benutzerdefinierten Filter an, die für eine spätere Verwendung gespeichert wurden.

Die Menuseite »Neue Dateien erstellen« führt in den drei Sektionen »Büro«, »System« und »Wissenschaft« Applikationen und Dateiformate auf:

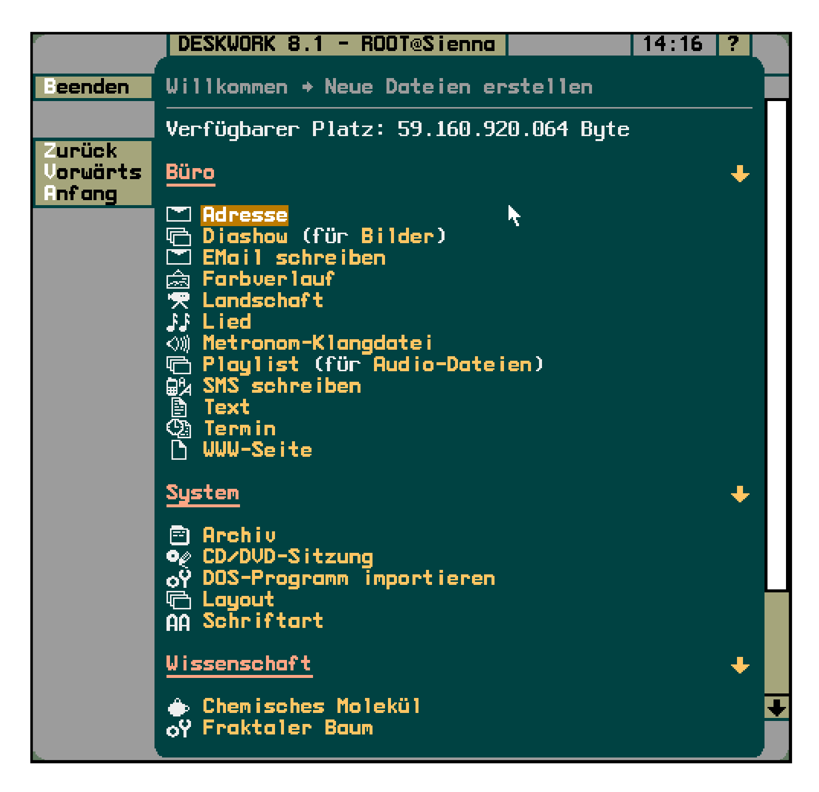


Abb. 7.1-9: Untermenu »Neue Datei erstellen«

Einige Menupunkte, wie z.B. »EMail schreiben«, sind hier aufgeführt, weil technisch betrachtet eine Datei vom Typ **dtMail** (→ *Kap. 6.2.1*) erzeugt wird; der Benutzer kann also erwarten, dass die entsprechende Applikation auf dieser Unterseite aufgeführt wird. Gleichzeitig ist das Verfassen von EMails eine wichtige Funktion heutiger Computersysteme, so dass diese Option noch einmal auf der Hauptseite in der Sektion »Nachrichten« angeboten wird (→ *Abb. 7.1-11*).

Die drei Sektionen »Büro«, »System« und »Wissenschaft« werden auf gleiche Weise auf der Menuseite »Vorhandene Dateien zeigen/ändern« (→ *Abb. 7.1-10*) benutzt; typbasierte Filter stellen also die primäre Zugriffsmöglichkeit auf die Library dar. Dieses Vorgehen erscheint besonders sinnvoll, da sich der Benutzer fast immer an den Typ einer gesuchten Datei wie »Audio-Datei« oder »Bild« erinnert.

Auf der Menuseite »Vorhandene Dateien zeigen/ändern« sind auch Dateitypen aufgeführt, die nicht vom Benutzer erzeugt, sondern vom Betriebssystem automatisch verwaltet werden. Darunter fallen z.B. Hilfethemen der Online-Hilfe oder Spiele-Highscores:

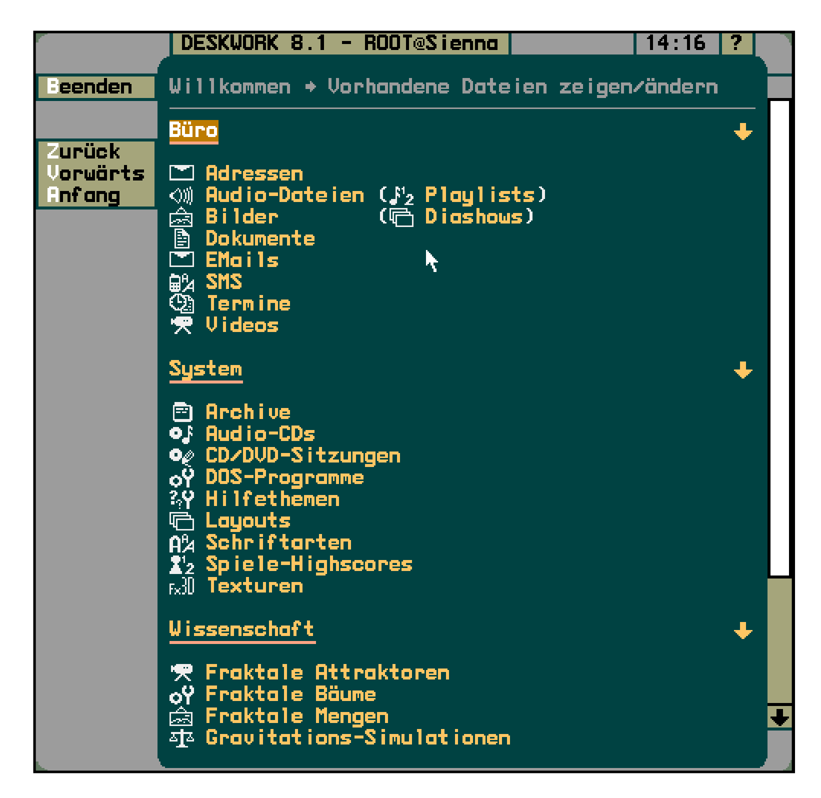


Abb. 7.1-10: Untermenu »Vorhandene Dateien zeigen und ändern«

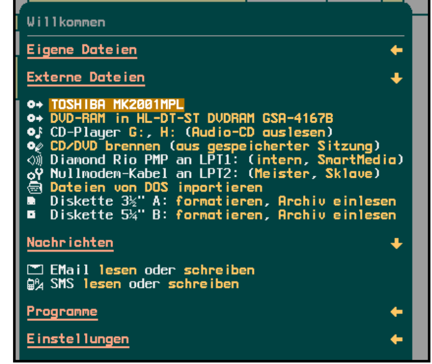
7.1.4 Vorteile

Hypertext-Menus können als Shell umfangreiche, nicht auf Verzeichnisse bzw. Dateinamen beschränkte Optionen für die Organisation von Dateien darstellen. Somit eignen sich derartige Shells für den Einsatz mit Libraries.

Darüber hinaus können Menuseiten abhängig von Ereignissen dynamisch erzeugt werden, etwa nach dem Einlegen von Datenträgern, beim Erkennen von defekten Festplatten, bei fälligen Terminen und vielem mehr. Auf diese Weise können Widgets ersetzt werden:



Nur statische Komponenten



Eingelegte PCMCIA-Festplatte und DVD-RAM



Alarme (»Ereignisse«)

Abb. 7.1-11: Dynamisch erzeugte Menuseite

In eine Menuseite eingebettete Hinweise sind wesentlich »diskreter« als die üblicherweise verwendeten Fenster zur Darstellung von Warnungen und Fehlermeldungen, da sie den Eingabefokus nicht blockieren. Der Benutzer wird daher nicht gezwungen, sofort auf Alarme zu reagieren.

Dies ist besonders vorteilhaft im Zusammenhang mit der »AutoPlay«-Funktion von Microsoft Windows. Beim Einlegen eines externen Datenträgers (CD, DVD, Flash-Speicherkarte, USB-Festplatte usw.) erscheint umgehend ein Menu, das dem Anwender diverse Aktionen präsentiert, die auf das eingelegte Medium angewendet werden können:

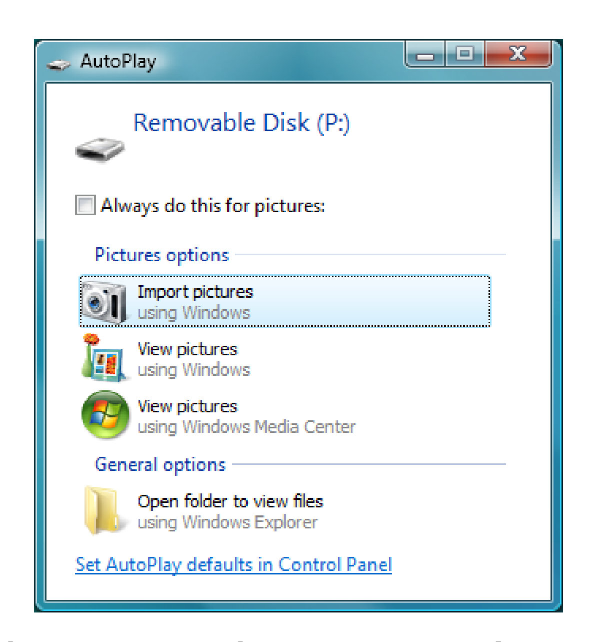


Abb. 7.1-12: »AutoPlay«-Menu von Windows Vista

Dieses Menu erscheint umgehend nach dem Mounten des Datenträgers, auch wenn der Benutzer zu diesem Zeitpunkt noch nicht mit den Dateien arbeiten oder sie direkt mit einem Programm öffnen möchte. Unter Umständen bietet das Menu die benötigten Funktionen gar nicht an; das Fenster stört also. Eine hypertextbasierte Shell verbessert diese Situation, indem alle gemounteten Laufwerke zusammen mit ihren jeweiligen Befehlen als Menueintrag präsentiert werden. Die Befehle können so auf der Hauptseite mit einem Klick aktiviert werden, ohne jedoch den Arbeitsfluss des Benutzers zu stören:

```
CD-Player G:, H: (Audio-CD auslesen)
CD/DVD brennen (aus gespeicherter Sitzung)
Diamond Rio PMP an LPT1: (intern, SmartMedia)
Nullmodem-Kabel an LPT2: (Meister, Sklave)
Dateien von DOS importieren
Diskette 3½" A: formatieren, Archiv einlesen
Diskette 5¼" B: formatieren, Archiv einlesen
```

Abb. 7.1-13: Gemountete Laufwerke und ihre Befehle

Während traditionelle Menus als Baum strukturiert sind, gestatten Hypertext-Seiten darüber hinaus weitere Strukturen, z.B. zirkuläre Verweise ohne Rückkehr auf eine übergeordnete Seite:

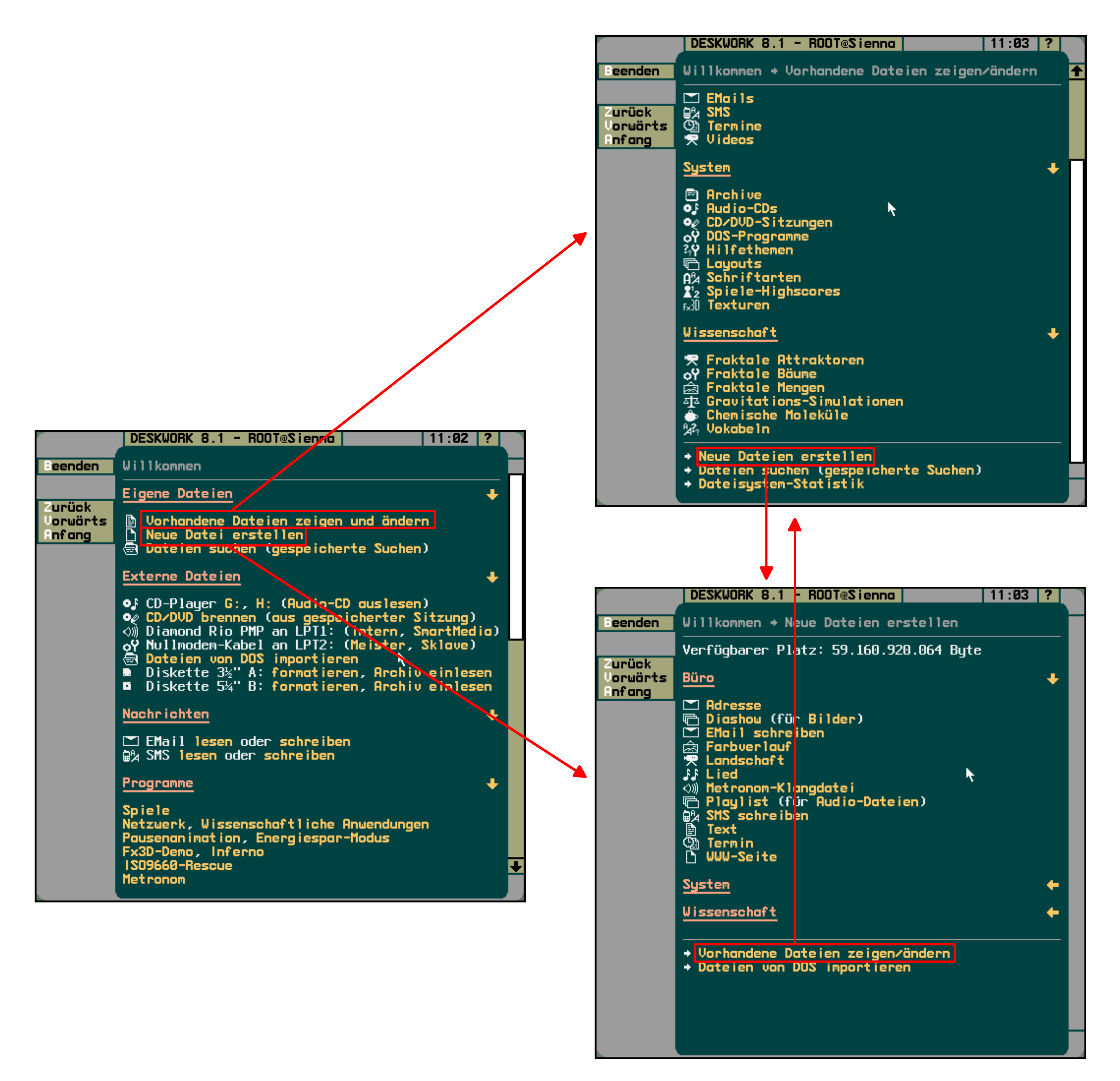


Abb. 7.1-14: Zirkuläre Verweise auf andere Menuseiten

7.2 Join-Operationen

Eine Beispiel für Suchfilter, die programmgesteuert erzeugt werden, sind Join-Operationen. Das Datenmodell einer Library (→ *Kap. 4.1*) bietet eine inhärente Unterstützung für Joins über beliebige Attribute, so dass Applikationen in unterschiedlichen Dateien gespeicherte Informationen verbinden können.

Diese Technik wertet Applikationen auf, indem etwa beim Betrachten eines Bildes Befehle zum Anzeigen anderer Bilder mit ähnlichen Eigenschaften (z.B. Aufnahmedatum oder Belichtung) angeboten werden, oder beim Abspielen von Musikdateien weitere Stücke desselben Künstlers zur Auswahl stehen. Als Fallbeispiel dient eine gespeicherte SMS, die neben dem eigentlichen Nachrichtentext auch die Telefonnummer der Absenderin enthält:

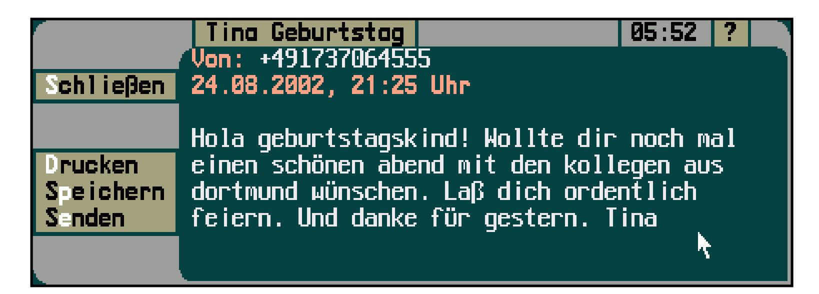


Abb. 7.2-1: SMS mit Telefonnummer der Absenderin

Die Referenz-Library kennt ein Dateiformat für Adressdateien (→ *Kap. 6.2.1*), die bei der Darstellung von SMS zur Auflösung der Telefonnummer benutzt werden. In → *Abb. 7.2-1* kann die Telefonnummer der Absenderin jedoch nicht aufgelöst werden, da der Join von SMS-Dateien und Adressen über die Attribute **Von** und **Mobiltelefon** keine Dateien liefern. Nun wird eine Adressdatei erstellt, in der die Telefonnummer, die in der obigen SMS enthalten ist, als **Mobiltelefon** eingetragen wird:

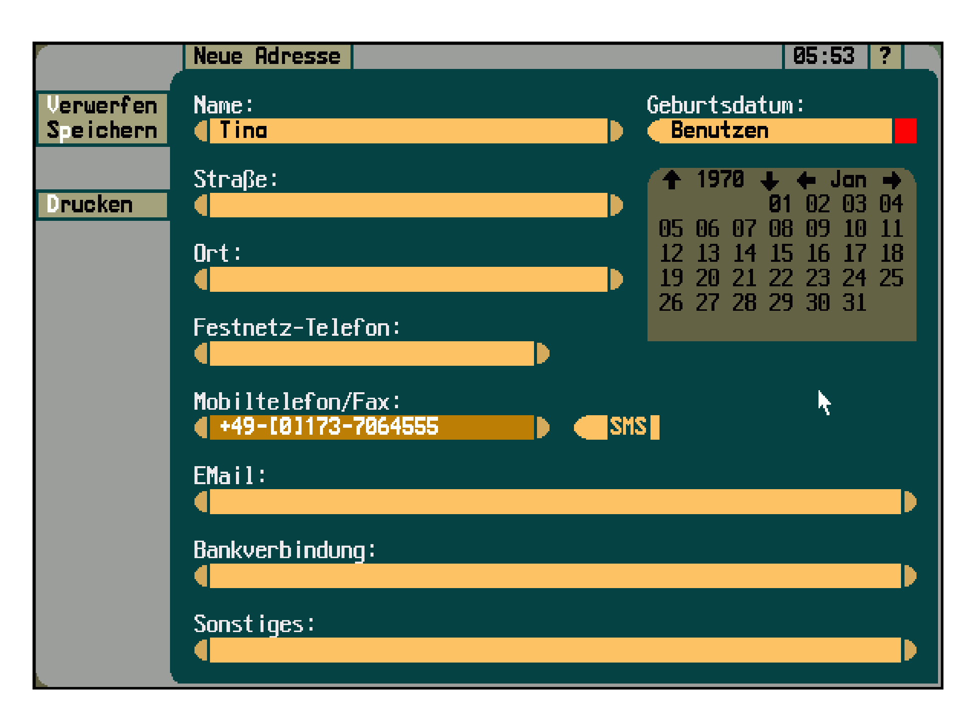


Abb. 7.2-2: Adresse, die eine Telefonnummer enthält

Wird nun die SMS noch einmal geöffnet, so erscheint neben der Telefonnummer auch der zugehörige Name, der durch eine Join-Operation über die Telefonnummer gefunden wurde:

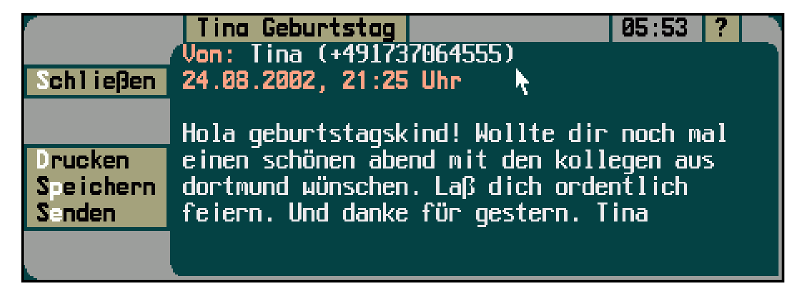


Abb. 7.2-3: Join von SMS und Adresse

7.3 Benutzerdefinierte Filter

Die in → *Kap. 7.1* vorgestellte Shell bietet einen Aufruf des Datei-Managers anhand des Dateiformats. Dieses Attribut wurde gewählt, weil sich Benutzer in der Regel an den Typ einer gewünschten Datei erinnern, wie etwa »Audio-Datei« oder »Bild«. Darüber hinaus ist es jedoch erforderlich, auch Suchanfragen bezüglich anderer oder mehrerer Attribute zu stellen, oder innerhalb des Datei-Managers ein Suchergebnis durch Hinzufügen weiterer Bedingungen zum Suchfilter (→ *Kap. 6.2.2*) zu verkleinern.

Zum Erstellen eines Suchfilters durch den Benutzer wurde ein Dialogfenster implementiert (→ *Abb. 7.3-1*). Mit der Schaltfläche »Hinzufügen« kann ein Attribut des globalen Datenraums ausgewählt werden, welches dann als Bedingung zum Filter hinzugefügt wird. Eventuell erforderliche Parameter dieser Bedingung können danach eingestellt werden. Mit den Schaltflächen am rechten Rand des Fensters werden einzelne Bedingungen wieder entfernt. Darüber hinaus kann der Anwender einen Suchbegriff zur Volltextsuche über alle Attribute eingeben, und Systemdateien vom Suchergebnis ausschließen:

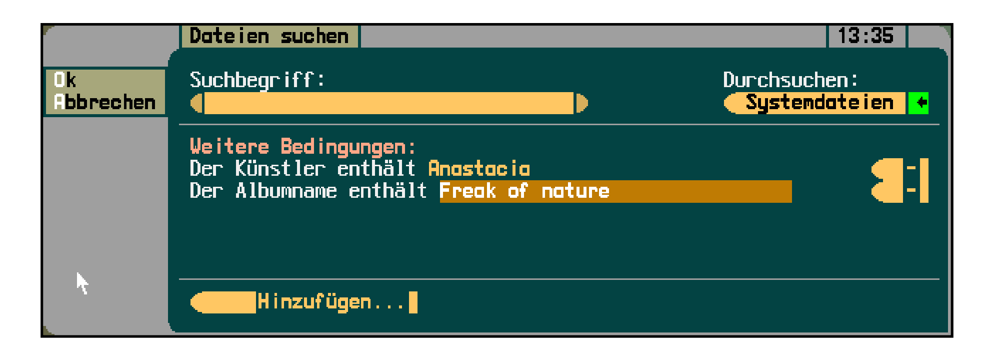


Abb. 7.3-1: »Datei suchen«-Dialog

7.3.1 Shell-Integration

Bei Suchfiltern handelt es sich um eine kompakte Datenstruktur (→ *Kap. 6.2.2*), die in einer Datei gespeichert werden kann. Daher wurde in der Referenz-Library ein neuer Dateityp **dtSearch=42** definiert (→ *Kap. 6.2.1*). Durch einen zusätzlichen Menupunkt in der Shell (→ *Abb. 7.3-2*) können alle Dateien des Typs **dtSearch** im Datei-Manager angezeigt werden. Dateien dieses Typs werden beim Ausführen vom Datei-Manager als neuer Suchfilter gesetzt, so dass sie dem Benutzer wie ein Unterverzeichnis erscheinen. Das zugehörige Applikations-Objekt (→ *Kap. 6.1.2*) gestattet das Modifizieren des Suchfilters.

→ Neue Dateien erstellen
→ Dateien suchen (gespeicherte Suchen)
→ Dateisystem-Statistik

Abb. 7.3-2: Gespeicherte Suchfilter in der Shell

7.3.2 Vorteile

Benutzerdefinierte Filter bilden die »intelligenten Wiedergabelisten« und Alben der Applikationen iTunes bzw. iPhoto nach, was sich auch an den jeweiligen Dialogfenstern zeigt:

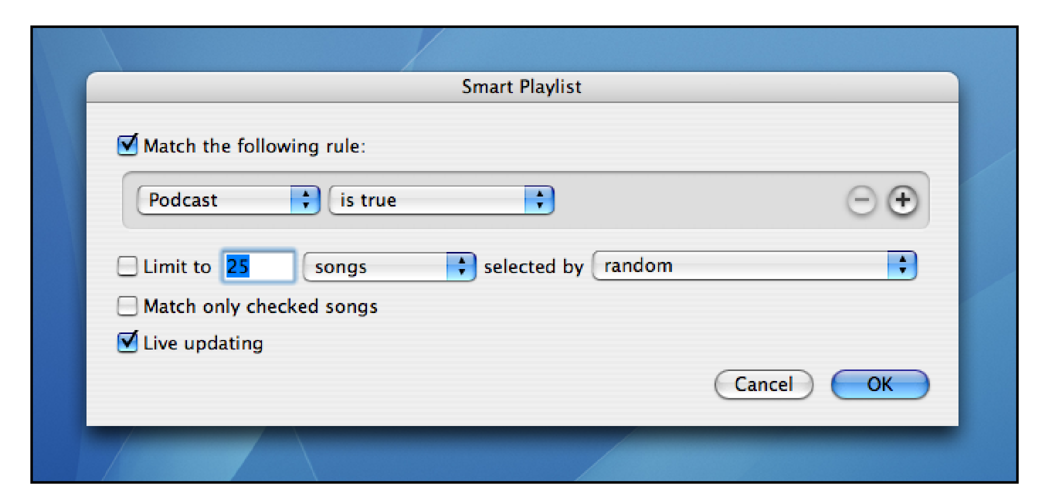


Abb. 7.3-3: »Intelligente Wiedergabeliste« in Apple iTunes

Die benutzerdefinierten Filter der Referenz-Library gehen jedoch über Wiedergabelisten hinaus. Durch die Einbindung der Library ins Betriebssystem (→ *Kap. 6*) können benutzerdefinierte Filter für beliebige Dateien definiert werden, und nicht nur für die Dateiformate einer bestimmten Applikation. Derartige Suchfilter sind also universell einsetzbar. Da sie beim Öffnen eine Suchanfrage an die Library auslösen, ist das von ihnen repräsentierte Suchergebnis darüber hinaus stets aktuell.

7.4 Automatische Ordner

Suchfilter sind im Idealfall so formuliert, dass alle relevanten Dateien gefunden werden (»recall«), aber keine unerwünschten Elemente enthalten sind (»precision«). Das erfordert einen genau formulierten Suchfilter, was für den Benutzer sehr aufwändig oder aufgrund der verfügbaren Attribute gar unmöglich ist.

[Nic06] und **[Mal83]** zeigen nun, dass ein Unterschied zwischen Suchen in den Bedeutungen von »browse« und »search« existiert. Demnach ist für den Menschen das Durchsuchen von bereits dargestellten Dateien wesentlich einfacher als das Erinnern an Eigenschaften. Es ist somit unabdingbar, Suchergebnisse im Datei-Manager sinnvoll aufzubereiten.

In einem ersten Schritt kann der Anwender eine Sortierreihenfolge für die dargestellten Dateien wählen, beispielsweise alphabetisch oder auf- bzw. absteigend nach Dateizeit oder Dateigröße. Dadurch werden beispielsweise aktuell in Arbeit befindliche Dateien weit oben gezeigt und damit schnell gefunden. Hat der Benutzer eine Sortierung ausgewählt, wird deshalb die Checkbox »Ordner bilden« aktiv:

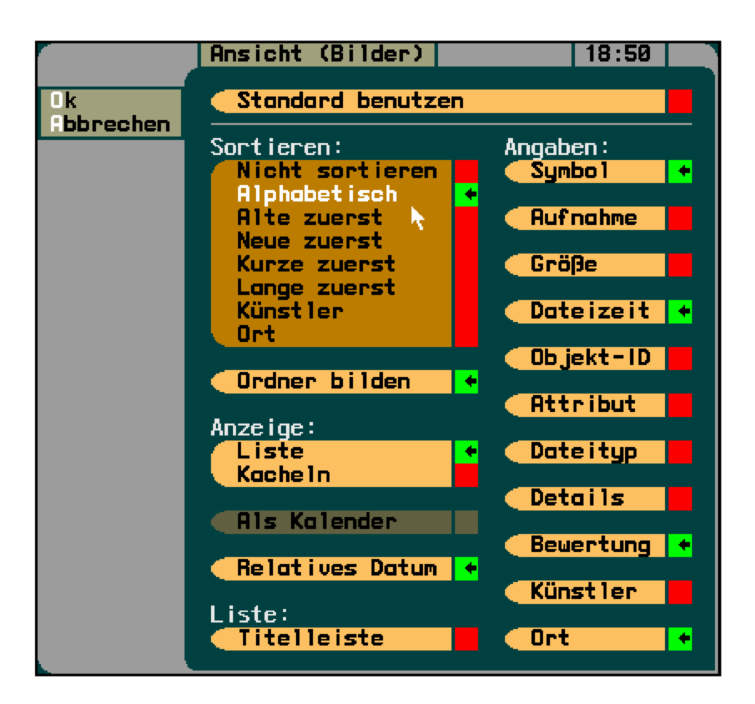


Abb. 7.4-1: Fenster »Ansicht« des Datei-Managers

Mit ihr werden in einem zweiten Schritt Dateien bezüglich des gewählten Sortierkriteriums gruppiert und als Unterverzeichnis dargestellt (→ *Abb. 7.4-3*):

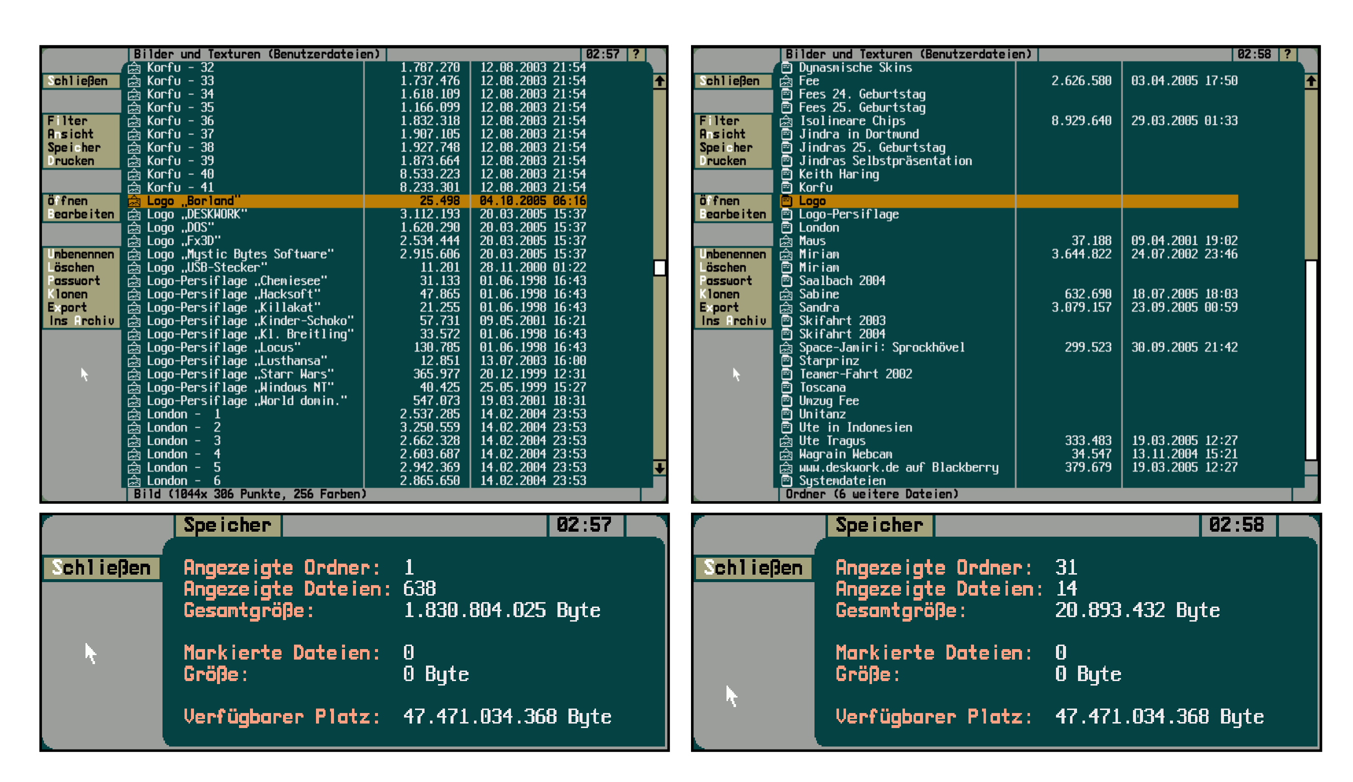


Abb. 7.4-2: 638 Bilder ohne Ordner

Abb. 7.4-3: 638 Bilder mit Ordnern

7.4.1 Kalenderansicht

Wenn automatische Ordner anhand von Tagen gebildet werden, ist eine Kalenderansicht für das Hauptverzeichnis verfügbar. Ein Klick auf einen markierten Tag öffnet sofort das entsprechende Unterverzeichnis, und zeigt somit alle Dateien eines bestimmten Tages:

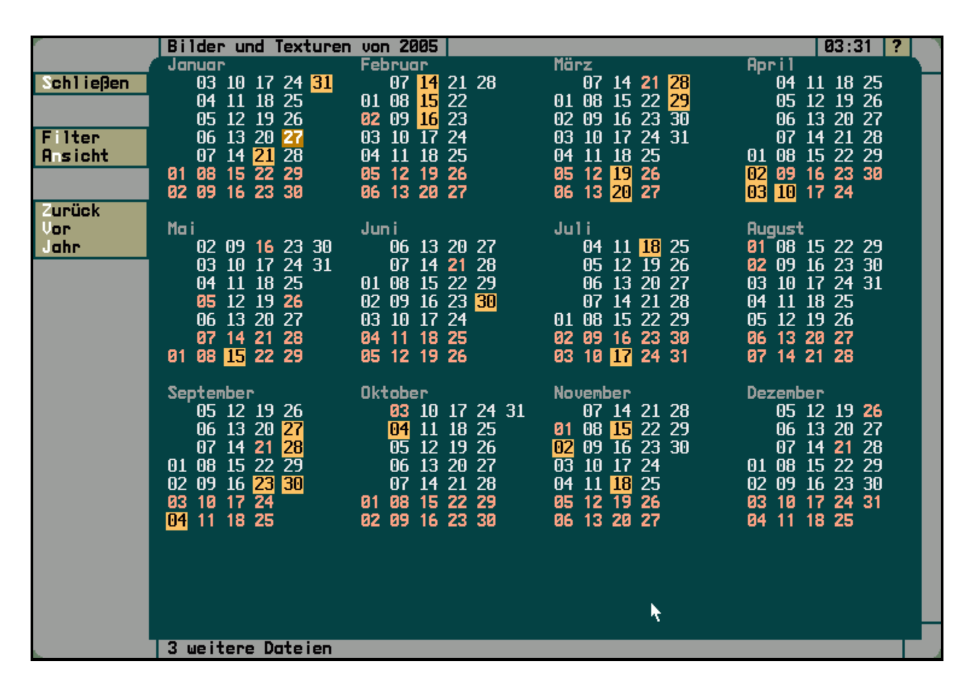


Abb. 7.4-4: Auf die Dateizeit bezogene Ordner, in Kalenderdarstellung

Automatische Ordner, und damit auch die Kalenderdarstellung, sind nur bei aktiver Sortierung verfügbar. Die Dateien im Suchergebnis werden vom Datei-Manager mittels Heapsort in $O(n \log n)$ sortiert, so dass bezüglich des Sortierkriteriums gleiche Dateien jeweils aufeinander folgen. Dadurch ist es in $O(n)$ möglich, solche Gruppen zu erfassen und durch einen Ordner zu ersetzen.

7.4.2 Vorteile

Automatische Ordner scheinen zunächst Verzeichnisse im herkömmlichen Sinn zu sein, die eigentlich vermieden werden sollen (→ *Kap. 1*). Es besteht jedoch ein großer Unterschied, denn diese Ordner werden **automatisch** anhand des Datenbestands gebildet. Daher sind automatische Ordner immer aktuell, zudem kann eine Datei nicht wie bei traditionellen Dateisystemen in einem falschen Ordner erscheinen. Wird beispielsweise eine Datei, die noch keinem Verzeichnis zugeordnet wurde, umbenannt, so erscheint sie ggf. sofort im entsprechenden Ordner.

Darüber hinaus stellen automatische Ordner kein starres Ordnungsschema für Dateien dar, denn die Gruppierung ändert sich sofort, wenn ein anderes Sortierkriterium ausgewählt wird. Es handelt sich also in der Terminologie von **[Mal83]** (→ *Kap. 1*) bei automatischen Ordnern um »piles«, die sich selbst ordnen bzw. neu zusammensetzen und so zu »files« werden – ein Vorgang, der bei Papierakten undenkbar ist. Zusammen mit einem Attribut, das das letzte Aufrufdatum einer Datei oder die

Anzahl der Aufrufe in einem bestimmten Zeitraum enthält, können »aktuelle« oder »wichtige« Dateien schnell aufgefunden werden.

Hiervon profitiert die in → *Kap. 7.1* vorgestellte Shell, die auf der Startseite u.a. alle fälligen Termine an prominenter Stelle anzeigt. Wird im Datei-Manager die Kalenderdarstellung auf Adressen oder Termine angewandt, so werden einschlägige Programme wie Terminplaner oder Adressverwaltungen mit Hilfe automatischer Ordner ersetzt.

7.5 Semantisches Tagging

In vielen Systemen, darunter DBFS (→ *Kap. 2.8*), können Dateien mit Schlagworten versehen werden, nach denen dann auch gesucht werden kann. Schlagworte stellen zur Zeit die beliebteste Form des *Taggings* dar **[Kre07]**. Die folgende Abbildung zeigt als weiteres Beispiel ein Foto innerhalb des Bilderdienstes Flickr **[Fli07]**, das mit den Tags **shdh**, **shdh15**, **jpf** und **nickpeters** versehen wurde:

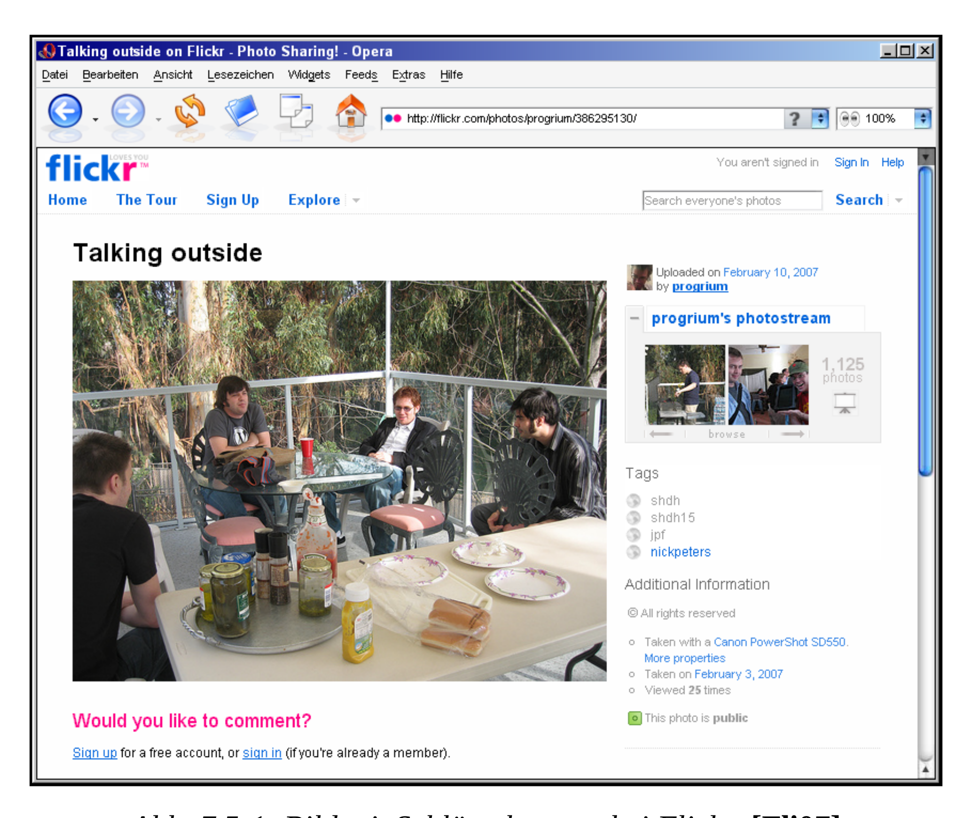


Abb. 7.5-1: Bild mit Schlüsselworten bei Flickr **[Fli07]**

Das Datenmodell einer Library (→ *Kap. 4.1*) gestattet beliebige Attribute innerhalb eines Objekt-Schemas, so dass Tagging durch Hinzufügen eines geeigneten Attributs zu allen Objekt-Schemata für alle Dateiformate unterstützt werden kann. Ebenso können auch mehrere Attribute mit unterschiedlichen Bedeutungen und Wertebereichen eingeführt werden, wodurch im Gegensatz zur einfachen Auflistung von Schlagworten ein »semantisches Tagging« implementiert wird. In diesem Abschnitt werden zwei in der Referenz-Library (→ *Kap. 6*) implementierte Tag-Typen vorgestellt.

7.5.1 Geotagging

Der Exif-Standard **[EXI07]** für die Einbettung von Metadaten in Bilddateien definiert zwei Attribute für GPS-Koordinaten. Von hochwertigen Kameras oder mit entsprechenden Zusatzgeräten werden die Koordinaten direkt im aufgenommenen Bild gespeichert und können später ausgewertet werden, etwa durch Darstellung auf einer Weltkarte. Diese Art des Geotaggings ist jedoch nur für Bilddateien verfügbar, und das Abspeichern der Aufnahmeposition ist nur mit hochwertigen Kameras oder Zusatzgeräten möglich **[Gro07]**. Darüber hinaus ist die Suche nach Fotos anhand von Längen- und Breitenangaben unbefriedigend. Dasselbe gilt für Ortsnamen, da diese sprachabhängig und nicht eindeutig sind (z.B. »München« und »Munich« bzw. mehrere Orte »Neustadt«).

Daher wird hier für das Geotagging die Verwendung von IATA-Flughafencodes **[IAT07]**, die Fluggästen von ihren Gepäckanhängern (engl. »baggage tag«) vertraut sind:



Abb. 7.5-2: Gepäckanhänger mit IATA-Codes

Schon 1948 haben sich die in der IATA **[IAT07]** zusammengeschlossenen großen Fluggesellschaften darauf verständigt, die flugrelevanten Abläufe des Geschäfts zu standardisieren. So kam es zum 3-Letter-Code-System. Darin sind beispielsweise in der ersten Kategorie weltweit alle von Fluggesellschaften angeflogenen Flughäfen mit einer 3-Buchstabenkennung enthalten. Versucht wurde, die jeweiligen Kennungen so zu kreieren, dass sie möglichst phonetisch an den dazugehörigen Ort erinnern und der erste Buchstabe des Codes mit dem vollständigen Ortsnamen übereinstimmt: FRA steht für Frankfurt, HAM für Hamburg oder MUC für München. Eine Ausnahme stellt beispielsweise der alte Flughafen von Dubai dar, der die Kennung DXB erhielt und damit einen Buchstaben führt, der überhaupt nicht im Namen »Dubai« enthalten ist. Der Grund ist einfach: es gab bereits einen 3-Letter-Code DUB, den Flughafen von Dublin. Der zuerst festgelegte Code bleibt grundsätzlich bestehen, um die Eindeutigkeit zu gewährleisten **[LHN03]**.

Für große Städte, die über mehrere Flughäfen verfügen, wurde zusätzlich noch ein übergeordneter sogenannter Metropolitan- oder City-Code eingeführt. So findet man unter dem City-Code **BER** für Berlin die Flughäfen **TXL** für Tegel, **THF** für Tempelhof und **SXF** für Schönefeld. Unter **LON** für London weist das System die dazugehörigen Flughäfen Gatwick (**LGW**), Heathrow (**LHR**), Luton (**LTN**), London City (**LCY**) oder Stansted (**STN**) aus. Ähnlich ist es in New York (**NYC**) mit den Flughäfen John-F.-Kennedy (**JFK**), La Guardia (**LGA**) und Newark (**EWR**), der sogar in einem anderen US-Bundesstaat liegt **[LHN03]**.

Weniger problematisch sind die Bezeichnungen der Orte, die in der zweiten Kategorie des 3-Letter-Code-Systems aufgeführt sind, obwohl sie nicht in direkter Verbindung zu einem Flughafen stehen. Es handelt sich um die Punkte, die mit einem Flugticket durch ein anderes Verkehrsmittel wie Bahn oder Bus erreicht werden. Der Kölner Hauptbahnhof hat beispielsweise die Kennung **QKL**, die Stadt Heidelberg als Busstation firmiert mit **HDB**. Unter der Kategorie drei sind Orte aufgelistet, die weder über einen Airport, noch über einen Bahnhof oder eine Busstation verfügen, aber dennoch wichtig für das Linienfluggeschäft sind. Darunter fallen beispielsweise die Standorte der Fluggesellschaften oder große Verkaufsstützpunkte **[LHN03]**.

Die Referenz-Library führt einen IATA-Code als Attribut in jedes Objekt-Schema ein (→ *Kap. 4.1*). Beim Erstellen eines Suchfilters (→ *7.3*) kann eine Ortsangabe als Bedingung hinzugefügt werden:

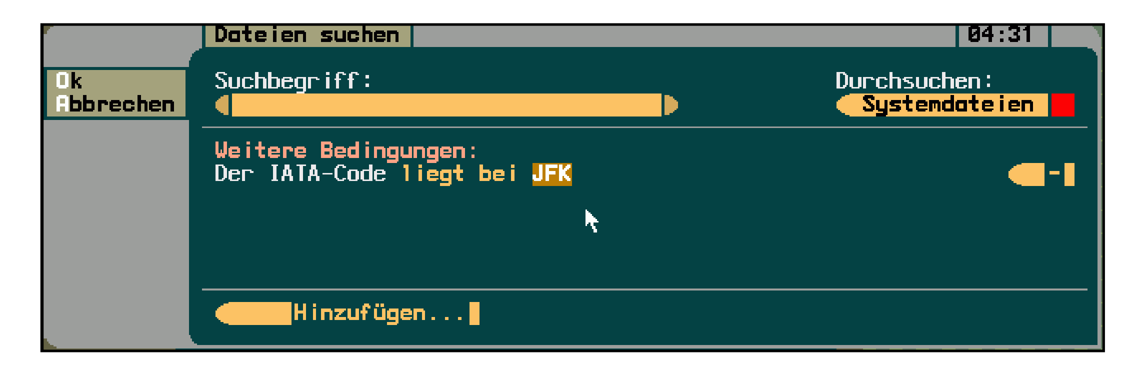


Abb. 7.5-3: Suche nach Dateien aus der Metropolitan-Area von JFK (= NYC)

Im obigen Beispiel ist als Modus »liegt bei« ausgewählt. Dadurch werden nicht nur alle Dateien ins Suchergebnis aufgenommen, die als Ortsangabe **JFK** (New York Kennedy-Airport) haben, sondern auch alle Dateien mit Orten, die in derselben Metropolitan-Area (hier **NYC**) liegen. Darunter fallen neben den anderen Verkehrsflughäfen auch diverse Orte in Manhattan, die Heliports besitzen (z.B. Hafen oder Wall Street).

Beim Erzeugen eines automatischen Ordners (→ *Kap. 7.4*) anhand der Ortangabe löst der Datei-Manager die IATA-Codes anhand einer internen Relation **IATACodes** zu Ortsnamen auf und fügt sie zum Ordnernamen hinzu. Für unterschiedliche Sprachversionen der Software können hier unterschiedliche Relationen mit den Ortsnamen in der jeweiligen Sprache benutzt werden (»München, Deutschland« bzw. »Munich, Germany«):

Bilder (Benutzerdateien) 04:33 ?
Schließen
Filter
Ansicht
Speicher
Drucken
Öffnen
Bearbeiten
BFE - Bielefeld, Deutschland
CFU - Korfu, Griechenland
DPS - Denpasar/Bali, Indonesien
DTM - Dortmund, Deutschland
DUS - Düsseldorf, Deutschland
EWR - Newark, USA
FLU - New York (Flushing), USA
FRA - Frankfurt, Deutschland
HAJ - Hannover, Deutschland
JOG - Jogjakarta, Indonesien
KLU - Klagenfurt, Österreich
LON - London, Großbritannien
MIL - Mailand, Italien
NRW - Nordrhein-Westfalen, Deu➔
NUE - Nürnberg, Deutschland
NYC - New York, USA
PEG - Perugia, Italien
QSF - San Francisco, USA
ROM - Rom, Italien
SAY - Siena, Italien
SFO - San Francisco (Internati➔
SJC - San Jose, USA
SZG - Salzburg, Österreich
UWP - Wuppertal, Deutschland
ZNB - Hamm, Deutschland
ZPE - Osnabrück, Deutschland
Ordner (12 weitere Dateien)

Abb. 7.5-4: Nach Orten gebildete automatische Ordner

Die Relation **IATACodes** wird nach Code sortiert gespeichert und enthält etwa 9.300 Einträge. Da die automatischen Ordner anhand dieses Attributs sortiert werden, können die dreibuchstabigen Ordnernamen durch einen Merge-Join in linearer Zeit mit Ortsnamen versehen werden. Die Laufzeit für das Erzeugen automatischer Ordner für 3-Letter-Codes ist also $O(n + |\textbf{IATACodes}|)$.

7.5.2 Bewertung von Dateien

Ein weiterer Tag-Typ der Referenz-Library (→ *Kap. 6*) hat als Wertebereich die Zahlen **1**, **2** und **3** und wird als Bewertung interpretiert. Ist der entsprechende Attributwert einer Datei **null**, so wird **0** bzw. »unbewertet« angenommen. Mittels dieses Attributs wird z.B. der Zugriff auf die Lieblingsmusik oder gern gesehene Filme enorm vereinfacht. In einem ersten Schritt wurden entsprechende Suchfilter in die Shell (→ *Kap. 7.1*) eingebunden (→ *Abb. 7.5-5*) und der Datei-Manager um die wahlweise Anzeige der Bewertung erweitert (→ *Abb. 7.5-6*). Da das Bewertungsattribut durch die Aufnahme in den Suchfilter (→ *Kap. 6.2.2*) systemweit zur Verfügung steht, profitieren davon auch andere Applikationen, wie etwa ein Mediacenter-Programm. In einer Kategorie »Favoriten« erscheinen auch die von der Library bereitgestellten virtuellen Abspiellisten, die automatisch alle MP3-Dateien mit mindestens einem, zwei oder drei Punkten enthalten und einfach angewählt werden können (→ *Abb. 7.5-7*).

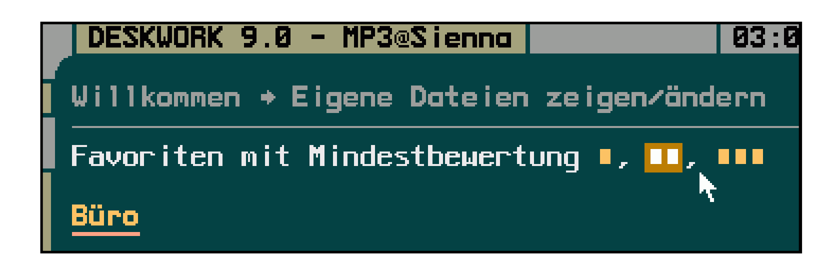


Abb. 7.5-5: Suchfilter für die Anzeige von Favoriten

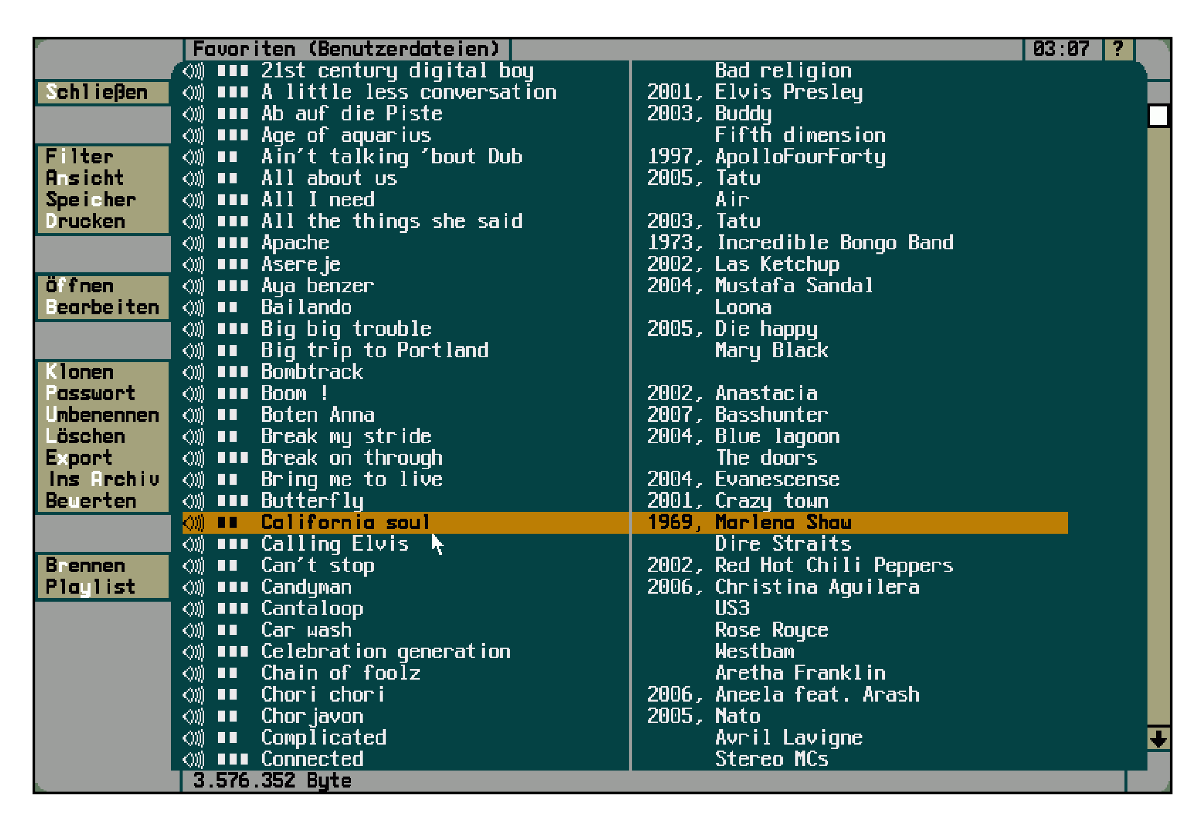


Abb. 7.5-6: Datei-Manager mit Favoriten (hier: mind. 2 Bewertungspunkte)



Abb. 7.5-7: Virtuelle Abspiellisten im »Medienzentrum«

7.5.3 Vorteile

Tagging erweitert die Metadaten, die von diversen Dateiformaten geliefert werden, um eine global gültige Kennzeichnung. Durch semantisches Tagging können Dateien nicht nur anhand von Schlagworten, sondern durch bestimmten Eigenschaften gesucht und kategorisiert werden. Tagging wird vom Library-Modell (→ *Kap. 4.1*) unterstützt, und steht daher systemweit für alle Dateiformate zur Verfügung.

Speziell für das Geotagging bieten die von der IATA **[IAT07]** eingeführten 3-Buchstaben-Codes Vorteile. Gegenüber GPS-Koordinaten sind sie leicht zu merken, da sie kurz sind und sich an natürliche Ortsnamen anlehnen. Dies gilt umso mehr, wenn die entsprechenden Orte tatsächlich besucht wurden, etwa beim Tagging von Urlaubsfotos und -videos, und das Tag vom Benutzer selbst vergeben wurde. Darüber hinaus sind sie genormt und dadurch eindeutig. Sie können bei der Bearbeitung einer Suchanfrage leicht verarbeitet werden, und sind vielen Menschen auch außerhalb der Luftfahrt-Branche vertraut:

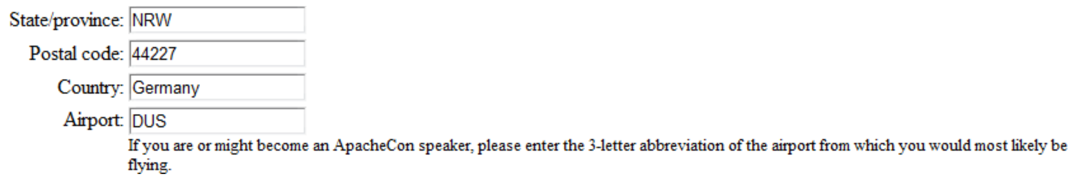


Abb. 7.5-8: Anmeldeformular für eine Konferenz (Ausschnitt)

Werden Tags manuell durch den Benutzer vergeben, so impliziert dies einen Aufwand, der mit dem Anlegen von Verzeichnissen vergleichbar ist (wobei allerdings eine Datei mehrere Tags erhalten kann). Semantisches Tagging kann jedoch auch automatisch durchgeführt werden, etwa durch Protokollieren aller Dateiaufrufe zur Ermittlung besonders häufig benutzter Dateien (→ *8.3.3*).

7.6 Aufgabenorientierung

Die einzelnen Objekt-Schemata der Referenz-Library (→ *Kap. 6*) wurden um ein zusätzliches Attribut erweitert, das alle für den Benutzer relevanten Methoden der jeweiligen Applikations-Klassen (→ *Kap. 6.1.2*) speichert (etwa »Weiterleiten« bei EMails). Auf diese Weise kann der Datei-Manager in Abhängigkeit von den gerade angezeigten bzw. markierten Dateien Befehle typspezifisch bereitstellen **[Dou00]**.

Die nachfolgenden Abbildungen zeigen die sog. »task pane« von Microsoft Windows XP. Hier werden unter dem Stichwort »Aufgaben« in Abhängigkeit von den markierten Dateien, etwa Bilder oder Videodateien, unterschiedliche Befehle angezeigt. Leider sind diese Befehle bei Windows XP lediglich in besonderen Verzeichnissen wie **Eigene Dateien\Eigene Bilder** aktiv, und das auch nur innerhalb des Explorers.



Abb. 7.6-1: Systemaufgaben von Windows XP *Abb. 7.6-2: Bildaufgaben von Windows XP*

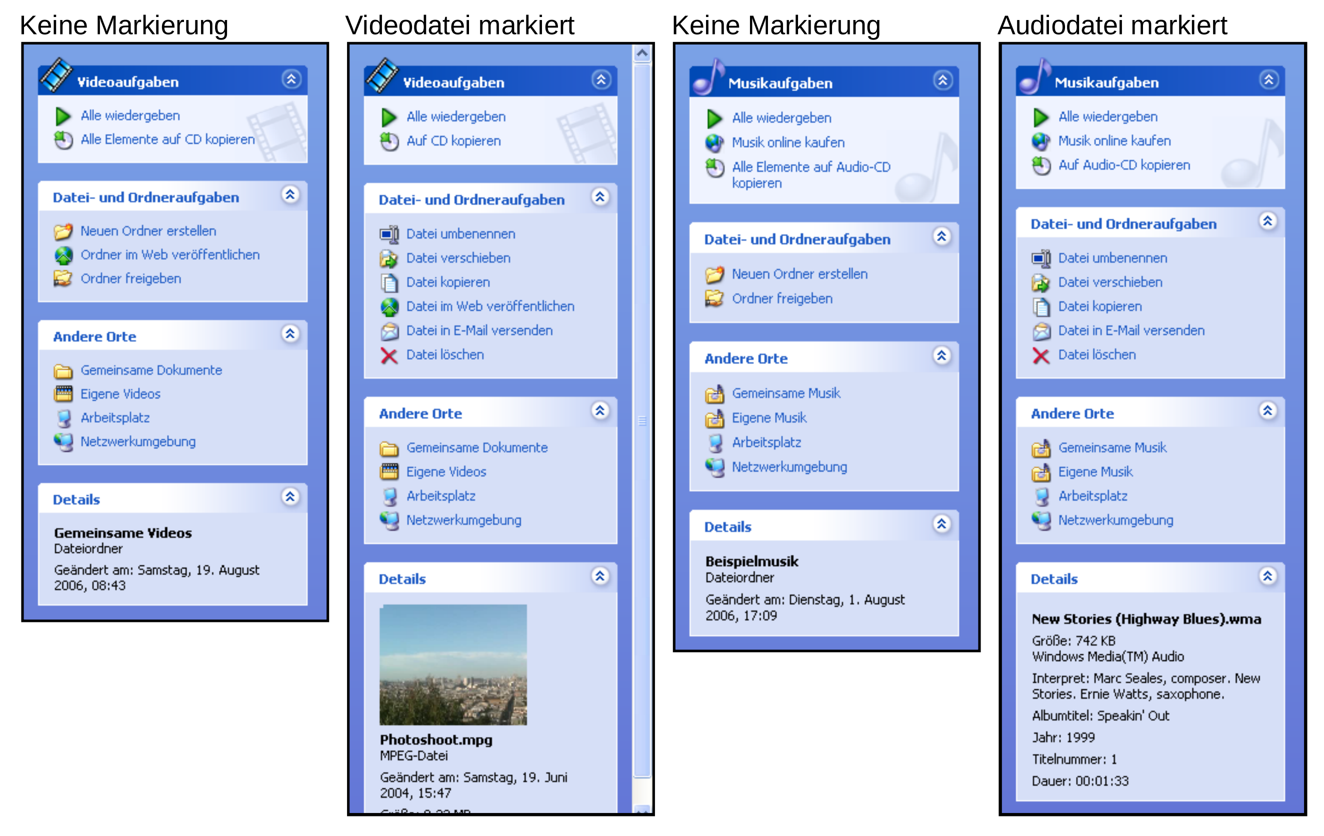


Abb. 7.6-3: Videoaufgaben von Windows XP *Abb. 7.6-4: Musikaufgaben von Windows XP*

Der Datei-Manager der Referenz-Library fragt die öffentlichen Methoden für alle markierten Dateiformate ab, und stellt die Schnittmenge der Befehlsmengen als Schaltflächen dar. In → *Abb. 7.6-5* zeigt die linke Spalte die Befehle »Öffnen« und »Bearbeiten«, die für einen automatischen Ordner (→ *Kap. 7.4*) verfügbar sind. Besteht die Auswahl aus Bildern, kommen weitere Befehle hinzu; »Diashow« ist beispielsweise nur für Bilder verfügbar und erzeugt mit dem entsprechenden Programm automatisch eine Präsentation aus allen markierten Bilddateien:

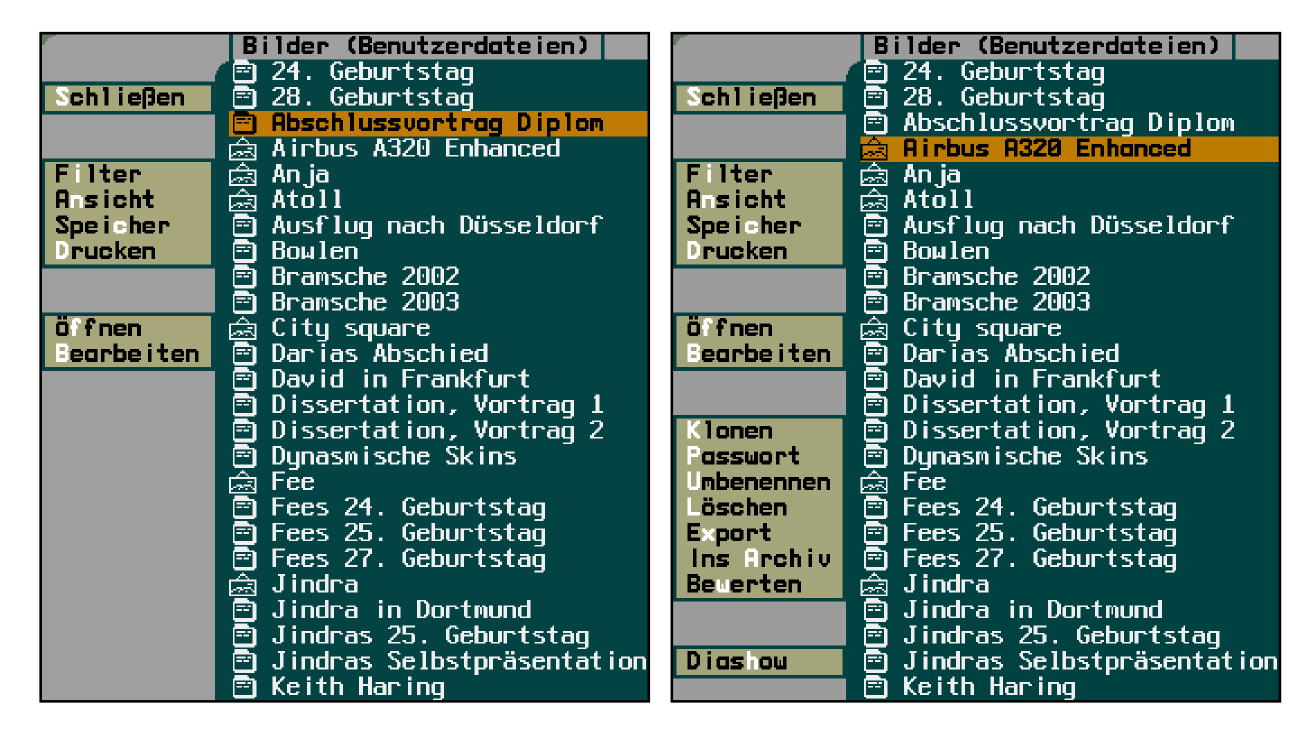


Abb. 7.6-5: Typspezifische Dateibefehle

7.6.1 Vorteile

Durch die Aufgabenorientierung werden dem Benutzer nur diejenigen Befehle gezeigt, die auf die gerade markierten Dateien anwendbar sind. Da in der Referenz-Library die für den Benutzer relevanten öffentlichen Methoden für alle Dateien bekannt sind, lässt sich die Aufgabenorientierung leichter umsetzen, und zwar einheitlich in allen Anwendungsprogrammen.

7.7 Webserver

Die Verwendung einer Library als Speichersystem impliziert Verbesserungen an anderen Programmen, die dies zunächst nicht vermuten lassen. Ein Beispiel hierfür sind »Webserver«, also diejenigen Programme, die eingehende HTTP-Anfragen beantworten. Die Verbesserungen an diesen Anwendungen werden in diesem Abschnitt u.a. anhand des sehr verbreiteten Apache HTTPD **[Apa07]** gezeigt. Wird über das HTTP-Protokoll eine URL angefordert, die keine Datei bezeichnet, sondern ein Unterverzeichnis, so wird im einfachsten Fall vom HTTPD eine Fehlermeldung zurückgesendet:

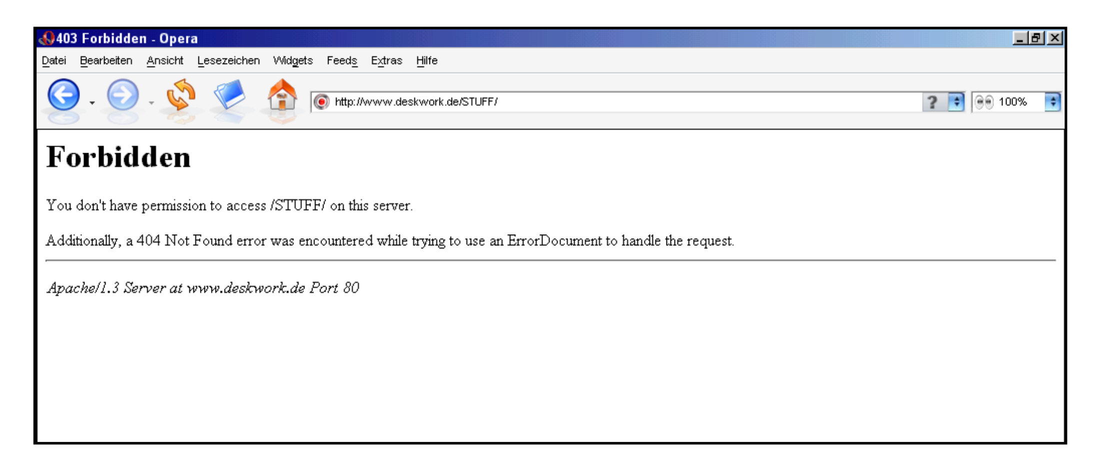


Abb. 7.7-1: Zugriff auf ein Verzeichnis ohne Indexierung durch den Webserver

Wenn der Administrator den HTTPD entsprechend konfiguriert hat, so wird – gleichsam als besonderer Service – vom Webserver **in Echtzeit** eine Liste aller im entsprechenden Ordner vorhandener Dateien und Unterverzeichnisse erzeugt und ausgeliefert:



Abb. 7.7-2: Zugriff auf ein Verzeichnis mit einfacher Indexierung durch den Webserver

Als Zusatzoption für die Indexierung von Verzeichnissen stellt der Apache HTTPD das »fancy indexing« bereit, bei dem die einfache Listendarstellung (»unordered list«, HTML-Tags <UL>…</UL>) durch eine Tabellendarstellung mit Icons und Zusatzinformationen ersetzt wird:

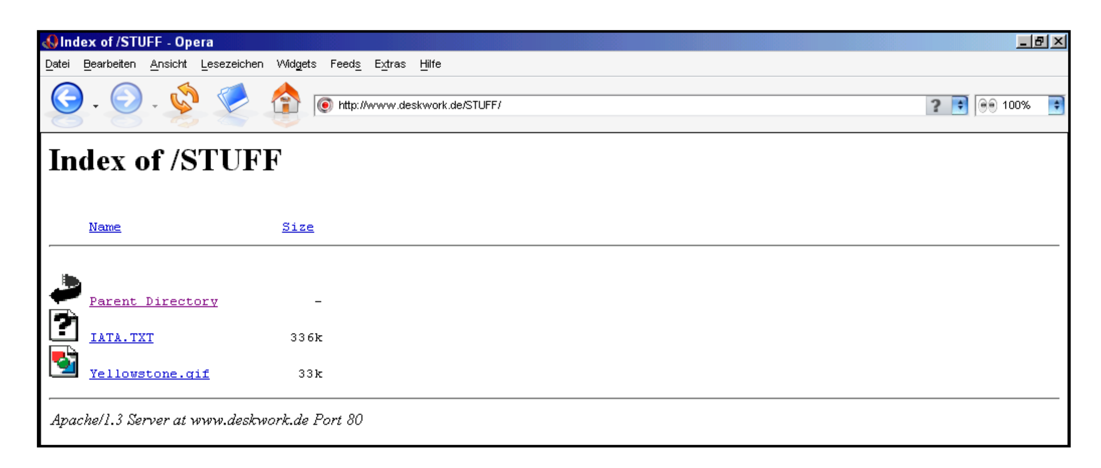


Abb. 7.7-3: Zugriff auf ein Verzeichnis mit »fancy indexing«

7.7.1 »Fancy fancy indexing«

Im Rahmen dieser Arbeit werden zwei Verbesserungen an der Echtzeit-Indexierung durch eine Webserver-Applikation vorgeschlagen, die in Anlehnung an die oben erwähnte Zusatzoption »fancy fancy indexing« genannt werden. Zur Demonstration wurde die Referenz-Library um eine Webserver-Applikation ergänzt **[Kol08d]**.

Neben einer modernen (d.h. auf CSS basierenden) Oberfläche ist es das Ziel, die von der Library verwalteten Metadaten auch beim Zugriff über HTTP nutzen zu können. Dies wird erreicht, indem vom Webserver in Echtzeit eine virtuelle Ordnerstruktur erzeugt wird, die sich von außen über URLs ansprechen lässt. Das Hauptverzeichnis / bietet analog zur Shell (→ *Kap. 7.1.3.1*) nach Dateitypen gegliederte Ordner an **[Kol08d]**:

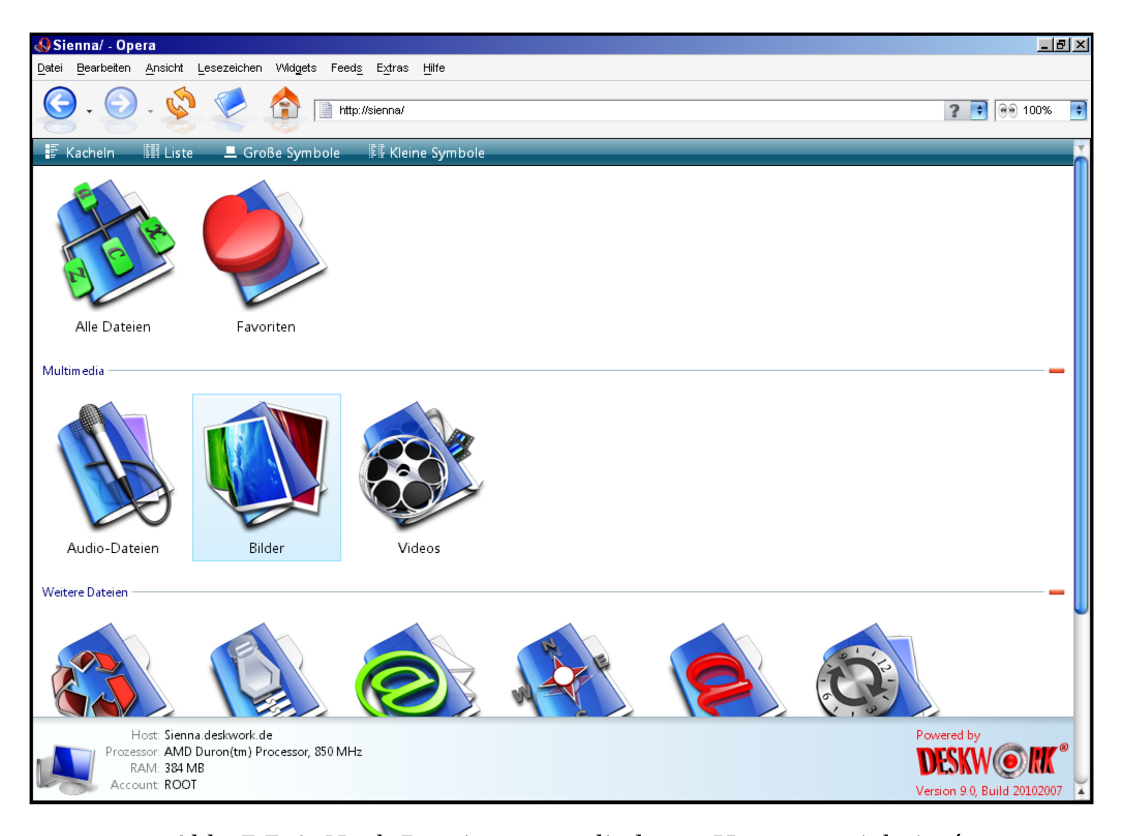


Abb. 7.7-4: Nach Dateitypen gegliedertes Hauptverzeichnis /

Bei Bildern werden zunächst alle Dateien vom Typ **dtDiashow** (→ *Kap. 6.2.1*) präsentiert, da diese mehrere Bilder zusammenfassen und daher von besonderer Bedeutung sind. Darunter können alle Einzelbilder nach verschiedenen Attributen gruppiert (→ *Kap. 7.4*) abgerufen werden:

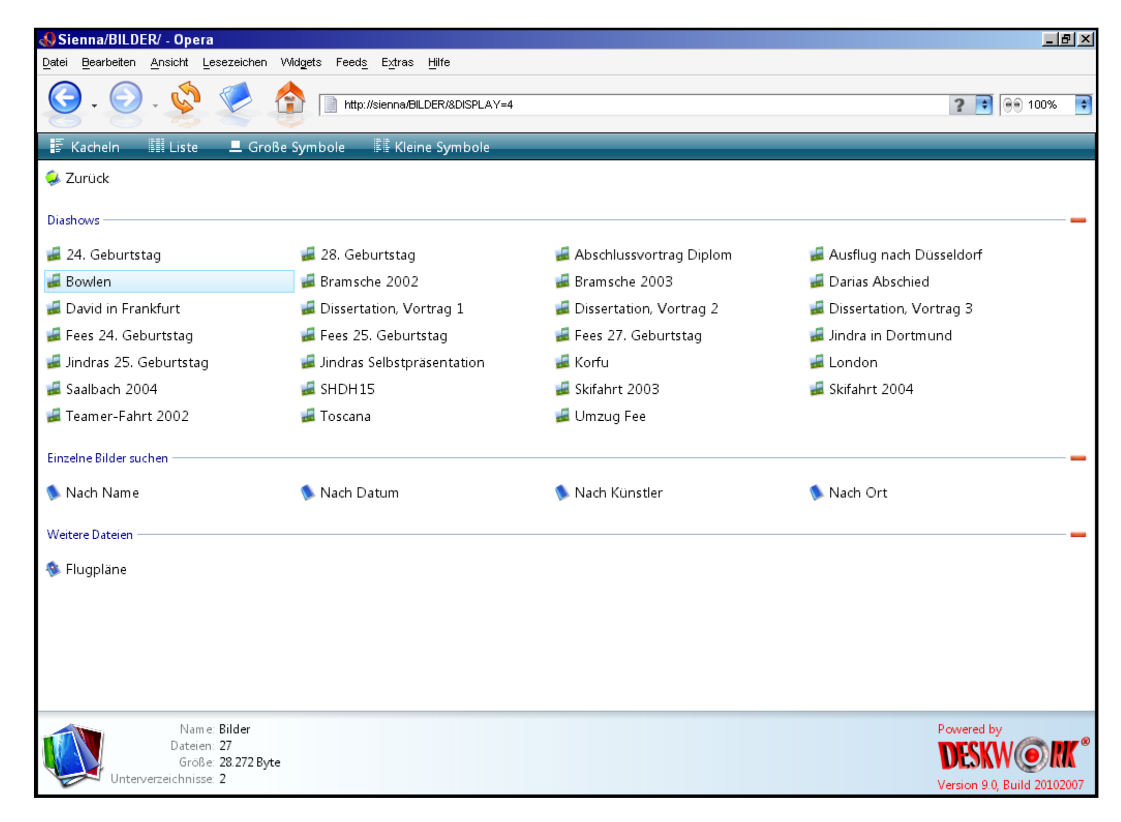


Abb. 7.7-5: */BILDER/*

Die folgende Abbildung zeigt alle Bilder nach Aufnahmeort (→ *Kap. 7.5.1*) sortiert. Der obere Abschnitt enthält automatische Ordner (→ *Kap. 7.4*) für alle Dateien mit Ortsangabe, darunter sind alle Dateien ohne Angabe dieses Attributs nach Dateiname sortiert, ggf. nach Dateinamen gruppiert:

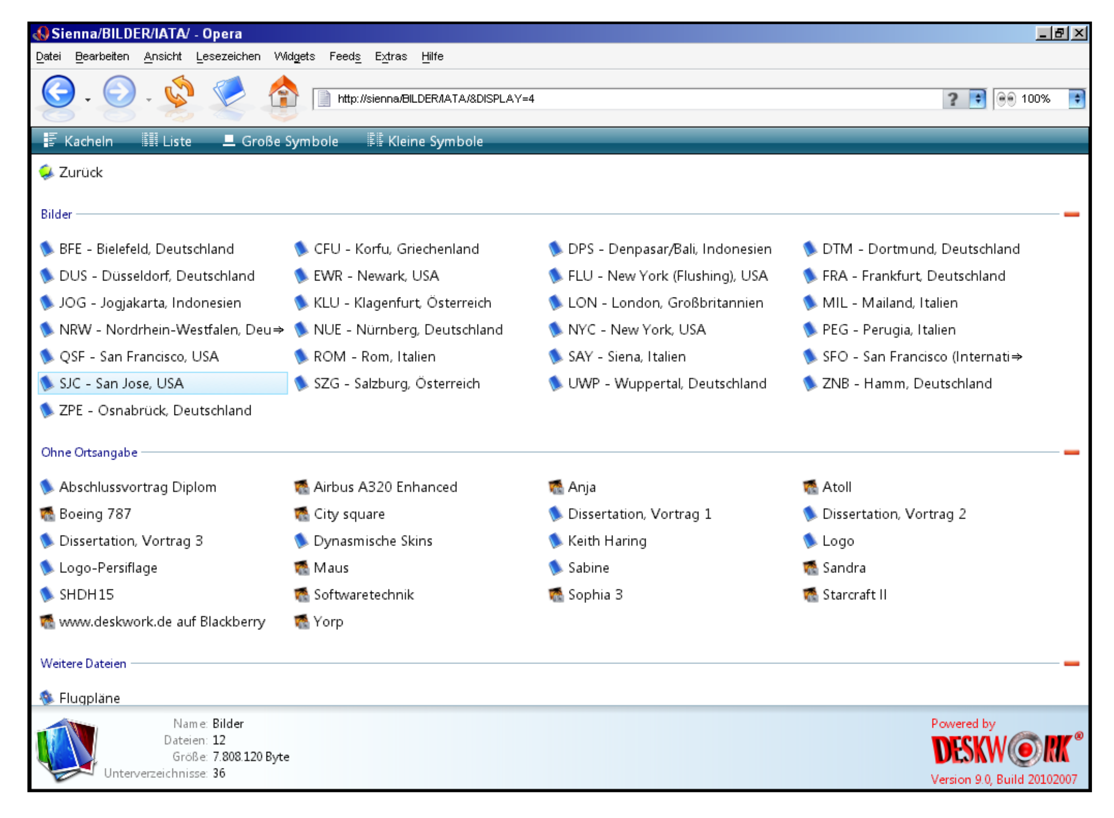


Abb. 7.7-6: */BILDER/IATA/*

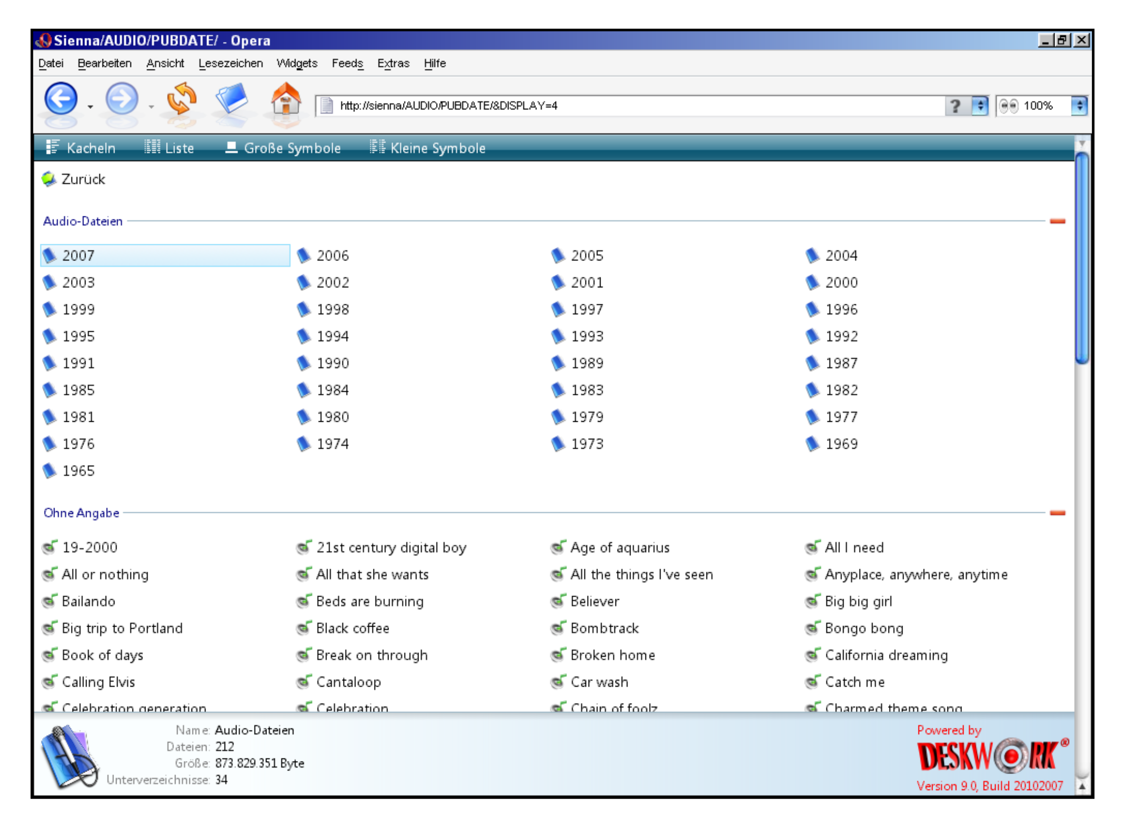


Abb. 7.7-7: */AUDIO/PUBDATE/*

Die obige Abbildung zeigt analog zu → *Abb. 7.7-6* alle Audio-Dateien, geordnet nach Erscheinungsjahr. Die untere Sektion präsentiert alle Dateien ohne entsprechende Angabe.

7.7.2 Vorteile

Mit »fancy fancy indexing« können Webserver-Applikationen für das automatische Indexieren von Dateien eine zeitgemäße, CSS-basierte Oberfläche anbieten. Mittels einer virtuellen Verzeichnisstruktur kann dabei auf den erweiterten Funktionsumfang einer Library zugegriffen werden.

In einer weiteren Ausbaustufe kann das »fancy fancy indexing« zur Einbindung von strukturierten Daten und Formularen genutzt werden, was die Erstellung von Web-Applikationen vereinfacht. Als Fallbeispiel dient ein System zur Verwaltung von Übungsaufgaben, den zugehörigen Lösungsvorschlägen von Studenten sowie den Bewertungen durch Dozenten. Die Hierarchie der einzelnen Datenobjekte (Vorlesungen enthalten Übungen, die wiederum Abgaben von Studenten enthalten) wird in diesem System auf virtuelle Verzeichnisse abgebildet. Besondere Sektionen der Webseite enthalten Übersichtstabellen und Eingabeformulare für Änderungen **[Kol08d]**:



Abb. 7.7-8: Hauptseite

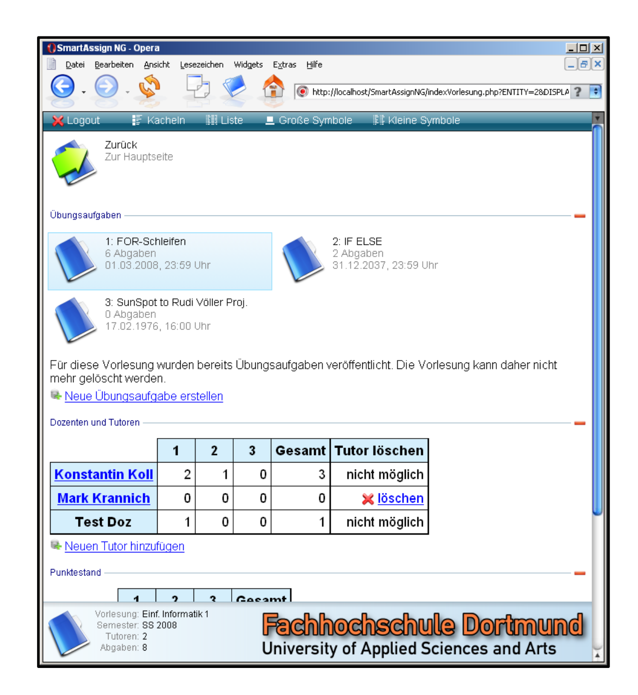


Abb. 7.7-9: Vorlesung (mit Tabellen)



Abb. 7.7-10: Aufgabe erstellen (Formular)

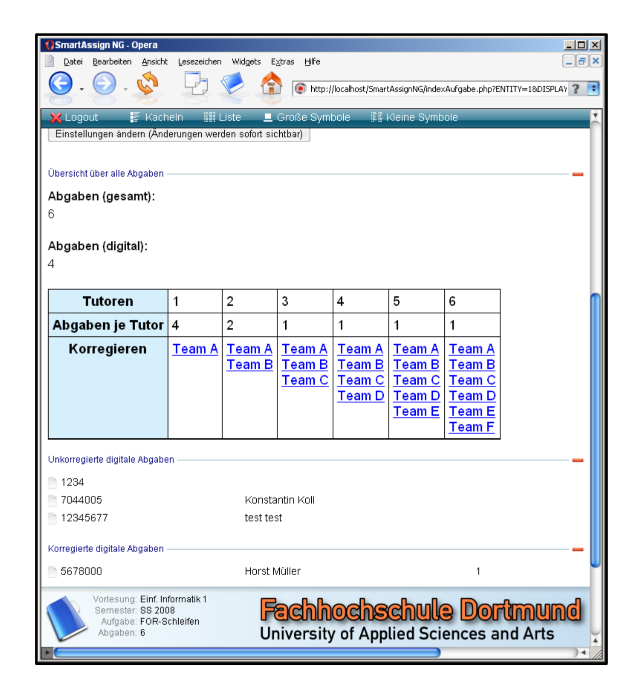


Abb. 7.7-11: Aufgabe (mit Tabelle und Liste)

8 Zusammenfassung und Ausblick

Zum Abschluss dieser Arbeit werden in diesem Kapitel die eingeführten Technologien und Verfahrensweisen zusammengefasst und bewertet (→ *Kap. 8.1*). Darüber hinaus wird der Einsatz in bereits existierenden Systemen diskutiert (→ *Kap. 8.2*), und es werden mögliche Weiterentwicklungen beschrieben (→ *Kap. 8.3*). Zuletzt wird mit »surface computing« ein Ausblick auf eine neue Art von Computersystemen gegeben, die vom Einsatz der vorgestellten Resultate profitieren kann (→ *Kap. 8.4*).

8.1 Zusammenfassung

Im Rahmen einer Bestandsaufnahme wurden unterschiedliche Ergänzungen traditioneller Dateisysteme untersucht, von einfachen Plug-Ins (→ *Kap. 2.6*) über Desktop-Suchmaschinen (→ *Kap. 2.10*) bis hin zu komplexen Weiterentwicklungen (→ *Kap. 2.7*). Einige Lösungen betten Dateien und ihre Applikationen dabei in eine objektorientierte Gesamtarchitektur ein. Insbesondere die heute verbreiteten Desktop-Suchmaschinen verändern jedoch nicht die Organisation des Dateisystems, sondern ermöglichen lediglich eine Suche nach Dateien über Verzeichnisgrenzen hinweg. Index-Methoden (→ *Kap. 3*), die entweder als Wortindex das Herkunftsattribut nicht speichern oder aber eine unzureichende Performanz haben, vermindern die Leistungsfähigkeit dieser Lösungen.

Zur grundlegenden Verbesserung der Organisationsstrukturen wurde das Datenmodell einer Library präsentiert (→ *Kap. 4.1*), das mächtig genug ist, um sowohl die Tupel relationaler Datenbanken als auch Dateien zu beschreiben; beide werden als gleichwertige Datenobjekte mit beliebigen Attributen gesehen und so integriert. Damit die Attributwerte aller Datenobjekte schnell zur Verfügung stehen, sollten sie indexiert werden. Zur Indexierung partiell belegter Datenräume (→ *Kap. 3.3.4*), die durch Tupel heterogener Schemata gebildet werden, eignet sich ein Master/Slave-Index (→ *Kap. 5*). Er unterstützt Partial Match Operationen (→ *Kap. 3.3.2*) und ist immun gegenüber degenerativen Effekten hochdimensionaler Datenräume (→ *Kap. 3.3.1*).

Auch bei Anwendung des Library-Modells können Dateien und Applikationen in eine objektorientierte Architektur eingebettet werden. Die in → *Kap. 6* vorgestellte Referenz-Architektur und ihre Implementierung liefern dafür eine Vorlage, welche die praktische Funktionsfähigkeit des Library-Modells nachweist. Die Referenz-Library wird außerdem eingesetzt, um die Performanz eines Master/Slave-Indexes zu messen (→ *Kap. 6.3*). Diese Indexierungsmethode ist anderen Ansätzen überlegen und bietet auch bezüglich des absoluten Zeitbedarfs eine ausreichende Leistung.

Eine Library integriert verschiedene Datenquellen, insbesondere relationale Datenbanken und klassische Dateisysteme, und verwaltet daher beliebige Attribute und Metadaten der jeweiligen Datenobjekte. Auf Basis eines derartigen Speichersystems konnte die Benutzerschnittstelle mittels diver-

ser Techniken, etwa Join-Operationen (→ *Kap. 7.2*) oder automatischer Ordner (→ *Kap. 7.4*), verbessert werden. Dadurch wird das ursprüngliche Ziel, den Zugriff auf Dateien zu optimieren (→ *Kap. 1*), erreicht.

8.2 Integration in existierende Systeme

Die in dieser Arbeit beschriebenen Verbesserungen lassen sich zum Teil in bereits bestehende Systeme integrieren. Dies gilt vor allem für den Master/Slave-Index (→ *Kap. 8.2.1*), aber auch für die Erweiterung vorhandener Dateisysteme (→ *Kap. 8.2.2*) und Applikationen (→ *Kap. 8.2.3*).

8.2.1 Master/Slave-Index

Am besten geeignet für den Einbau in existierende Systeme ist der Master/Slave-Index (→ *Kap. 5*). DBMS bieten häufig mehrere Datenstrukturen zur Indexierung an (→ *Kap. 3.3*), so dass ein Master/Slave-Index (ggf. in einer fortschrittlicheren Variante, → *Kap. 8.3.1*) als zusätzliche Methode hinzugefügt werden kann. Das DBMS wird dadurch in die Lage versetzt, hochdimensionale Datenobjekte mit heterogenen Schemata effizient zu indexieren. Ein entsprechend ausgerüstetes DBMS kann für alle Indexe über mehr als d Attribute automatisch einen Master/Slave-Index als Datenstruktur auswählen, oder überlässt den jeweiligen Anwendungsprogrammen die Wahl durch eine geeignete SQL-Erweiterung (z.B. `CREATE MASTERSLAVE INDEX idx ON table`).

Auf diese Weise kann eine Vielzahl von Anwendungen und Produkten, die entsprechend ausgerüstete DBMS einsetzen, profitieren. Darunter befinden sich nicht nur Desktop-Suchmaschinen und Systeme wie Microsoft WinFS (→ *Kap. 2.9*), das auf dem Microsoft SQL Server (→ *Kap. 3.3.5.1*) basiert, sondern beispielsweise auch Data Warehouses.

Darüber hinaus lassen sich die Operationen auf dem Master/Slave-Index ressourcenschonend implementieren, was die Datenstruktur auch für mobile Systeme, etwa Smartphones und MP3-Player, geeignet macht.

8.2.2 Erweiterung für Dateisysteme

Weitaus komplexer als die Integration des Master/Slave-Indexes in Datenbanksysteme ist die Integration des Library-Modells (→ *Kap. 4.1*) in traditionelle Datei- und Betriebssysteme. Ähnlich wie bei der Referenz-Library (→ *Kap. 6*) kann das bereits vorhandene Dateisystem als Grundlage für eine Library dienen, die parallel zum eigentlichen Dateisystem das Speichern von Dateien ermöglicht. Ein zusätzlicher Datei-Manager und speziell ausgestattete Applikationen können dann über eine zusätzliche Programmierschnittstelle auf die Library zugreifen.

Dies ist jedoch unbefriedigend, da die Library ins normale Dateisystem integriert werden soll. Dazu existieren je nach Plattform unterschiedliche Möglichkeiten. Die besten Voraussetzungen bieten Unix-Betriebssysteme, da der globale Namensraum für Dateien über Mountpoints (→ *Kap. 2.2*) ohnehin aus mehreren Dateisystemen zusammengesetzt wird. Über einen solchen Mountpoint können die in einer Library gespeicherten Dateien ins Unix-Dateisystem eingebunden und für Applikationen, welche die Programmierschnittstelle der Library nicht nutzen, zugänglich gemacht werden. Darüber hinaus können bei vielen Desktop-Managern, darunter KDE **[KDE06]**, die Standard-Dialoge zum Öffnen und Speichern von Dateien ersetzt werden (→ *Kap. 2.8*), so dass auch Legacy-Software die Möglichkeiten einer Library eingeschränkt nutzen kann.

Ein ähnlicher Mechanismus ist auch für die Windows-Plattform verfügbar, und wird beispielsweise von Microsoft WinFS (→ *Kap. 2.9*) und von Gerätetreibern (→ *Abb. 8.2-1*) genutzt. Mittels »Explorer Namespace Extensions« können innerhalb der Verzeichnishierarchie Mountpoints erstellt werden, deren weitere Unterverzeichnisse und Dateien von Plug-Ins ins Dateisystem abgebildet werden **[Mic08c]**. Auf diese Weise integriert Microsoft WinFS seine »Stores« ins Dateisystem (→ *Abb. 2.9-3*), aber auch die Treiber von mobilen Geräten ermöglichen dem Benutzer auf diese Weise den Zugriff auf die dort abgespeicherten Informationen:

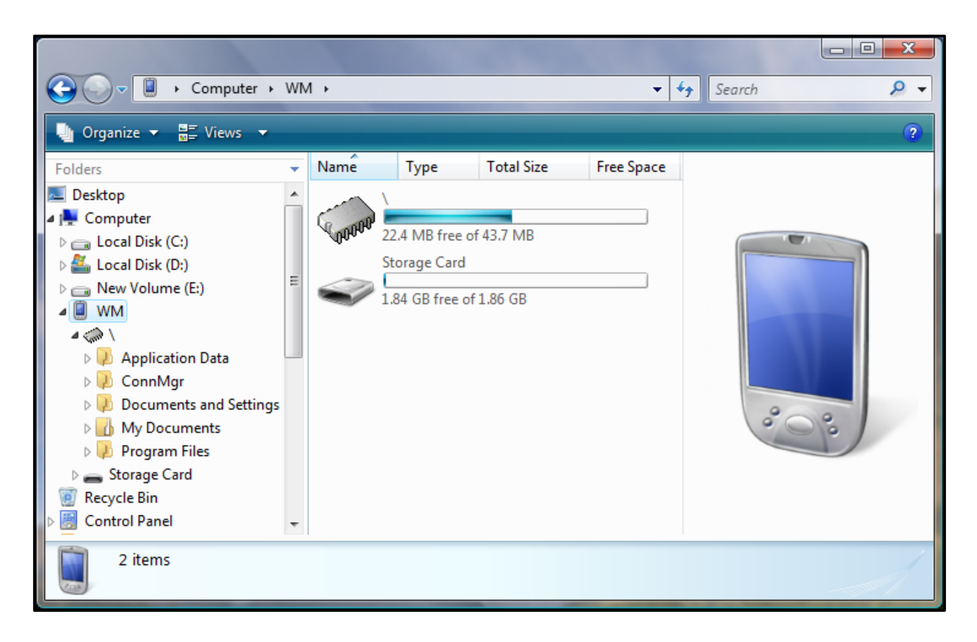


Abb. 8.2-1: Virtueller Namensraum unter Microsoft Windows

Analog zu Microsoft WinFS kann dieser Mechanismus genutzt werden, um die Dateien einer Library für existierende Software zugänglich zu machen.

8.2.3 Applikationen

Einige der in → *Kap. 7* präsentierten Verfahrensweisen können unabhängig vom Einsatz einer Library bereits existierende Applikationen aufwerten. So wird zur Zeit ein Modul für den Apache HTTPD entwickelt, der die übliche Indexierung von Verzeichnissen durch »fancy fancy indexing«

(→ *Kap. 7.7*), also einer CSS-basierten Benutzeroberfläche, ersetzen soll **[Kol08d]**. Eine Web-Applikation zur Abgabe und Bewertung von Übungsaufgaben, die ebenfalls diese Technologie nutzt, befindet sich als Showcase-Anwendung seit März 2008 an der Fachhochschule Dortmund im dauerhaften Einsatz (→ *Kap. 7.7.2*).

Auch Datei-Manager für traditionelle Dateisysteme, die ohne die Funktionalität einer Library auskommen müssen, können von einigen Verbesserungen profitieren: unter anderem können automatische Ordner anhand der vom Dateisystem verwalteten Attribute gebildet werden, was in der Regel eine Kalenderansicht (→ *Kap. 7.4*) ermöglicht.

8.3 Weiterentwicklungen

Die in dieser Arbeit eingeführten Technologien bieten ein erhebliches Anwendungspotential. In diesem Abschnitt werden daher zukünftige Entwicklungsschritte präsentiert, deren Umsetzung in naher Zukunft realistisch erscheint.

8.3.1 Master/Slave-Index

Der in → *Kap. 5* eingeführte Master/Slave-Index kann als zusätzliche Methode in bereits existierende DBMS integriert werden (→ *Kap. 8.2.1*). Um den Anforderungen kommerzieller Datenbanksysteme gerecht zu werden, müssen die Algorithmen, die auf einem Master/Slave-Index operieren, um die Fähigkeit der nebenläufigen Ausführung (»concurrent access«) erweitert werden.

Da Suchvorgänge keine Veränderungen am Index vornehmen (→ *Kap. 5.1.2*), können diese bereits jetzt nebenläufig durchgeführt werden. Das Hinzufügen von Tupeln muss serialisiert werden, da neue Daten immer ans Ende der Indextabellen angehängt werden (→ *Kap. 5.1.3*); möglicherweise gleichzeitig ablaufende Suchvorgänge müssen am alten Dateiende angehalten oder beendet werden, bis das Hinzufügen abgeschlossen ist. Sollen Einträge geändert oder gelöscht werden, müssen Suchvorgänge unter Umständen neu gestartet werden, damit keine inkonsistenten Suchergebnisse an den Aufrufer übermittelt werden.

Darüber hinaus bleibt zu untersuchen, ob andere Datenstrukturen als Slave-Indexe eingesetzt werden können und auch sinnvoll sind. Denkbar ist beispielsweise ein Wortindex auf Basis von invertierten Listen für Textdokumente, um ihren Inhalt zu indexieren. Ein derartiger Index könnte auch zusätzlich zu einem regulären Slave-Index für strukturierte Attribute eingesetzt werden, so dass sich die Informationen von Textdateien auf drei Indexe verteilen.

8.3.2 Semantisches Tagging

Durch semantisches Tagging werden statt einer Liste mit Schlüsselworten unterschiedliche Typen von Tags verwaltet; implementiert wurden bisher Ortsangaben (→ *Kap. 7.5.1*) und eine Bewertung von Dateien (→ *Kap. 7.5.2*). Darüber hinaus sind weitere Tags denkbar, die teilweise sogar schon in anderen Umgebungen verfügbar sind. Ein Beispiel ist das Taggen von Bilddateien mit Informationen über dargestellte Personen, wie es etwa die Community-Website »StudiVZ« **[Stu07]** anbietet:



Abb. 8.3-1: Tagging mit Personendaten **[Stu07]**

Ein Personen-Tag bei StudiVZ besitzt mehrere Attribute, u.a. den Namen und die Hochschule der Person und, da hier nur Bilder getaggt werden können, auch eine *X*- und *Y*-Koordinate innerhalb des Bildes. Die Informationen, die zur Person selbst gespeichert werden, können um beliebige Attribute wie beispielsweise Anschrift oder Geburtsdatum ergänzt werden, und enthalten im Extremfall eine vollständige Adressdatei **[RFC2425]** bzw. einen Verweis auf eine solche.

Ein anderer Tag-Typ kann für Termindateien **[RFC2445]** vorgesehen werden. Auf diese Weise können beliebige Dateien mit einem Terminplaner verknüpft werden, um Dateien mit dem zugehörigen Kalendereintrag zu verbinden, oder um Dateien zur Wiedervorlage zu markieren.

8.3.3 Automatisches Tagging

Tagging erweitert die Objekt-Schemata einer Library um zusätzliche Informationen, nach denen Dateien gesucht und mittels automatischer Ordner (→ *Kap. 7.4*) auch gruppiert werden können. Diese Vorteile werden durch einen gewissen Aufwand des Benutzers erkauft, der eigentlich ver-

mieden werden soll (→ *Kap. 1*). Aus diesem Grund ist die Automatisierung des Taggings anzustreben.

Für die beiden in → *Kap. 7.5* vorgestellten Arten von Tags ist diese leicht durchführbar: beispielsweise können die von hochwertigen Kameras in Bilddateien gespeicherten GPS-Koordinaten mittels einer internen Tabelle um den nächstgelegenen IATA-Code (→ *Kap. 7.5.1*) ergänzt werden. Für die Bewertung von Dateien ist eine Vergabe anhand einer Aufrufstatistik möglich, so dass beispielsweise die 10% am häufigsten geöffneten Dateien automatisch mit 3 Punkten, die nächsten 15% mit 2 Punkten, und die folgenden 25% mit 1 Punkt angezeigt werden.

Auch für weitere semantische Tags (→ *Kap. 8.3.2*) sind Automatisierungen durchführbar: so können mittels bilderkennender Verfahren und in Adressdateien gespeicherter Referenzbilder automatisch Personen in Bildern erkannt und getaggt werden. Beim Erstellen oder Speichern von Dateien können Termindateien automatisch als Tag hinzugefügt werden, wenn der Zeitpunkt des Speicherns in den Zeitraum des Termins fällt. Auch ein Join (→ *Kap. 7.2*) über die Dateizeit ist denkbar.

8.3.4 Verbesserte Visualisierung

Traditionelle Datei-Manager stellen die Dateien eines Verzeichnisses bzw. eines Suchergebnisses als Liste mit mehr oder weniger Angaben je Datei dar. Durch automatische Ordner (→ *Kap. 7.4*), die Dateien anhand eines vorher gewählten Attributs gruppieren, wird eine Darstellung als Kalender ermöglicht (→ *Abb. 7.4-4*).

Darüber hinaus sind weitere Varianten zur Visualisierung von Dateien denkbar, die allerdings aufgrund der beschränkten Möglichkeiten der zur Implementierung genutzten Plattform nicht realisiert werden konnten. Ein Beispiel hierfür liefert die Website »Tag Galaxy« **[Tag08]**, die anhand eines eingegebenen Begriffs Fotos vom Bilderdienst »Flickr« **[Fli07]** lädt und als Ball darstellt. Dieser Ball eignet sich, um schnell einen Überblick über eine größere Anzahl Bilder oder auch Videos zu bekommen; er kann mit der Maus beliebig gedreht werden:



Abb. 8.3-2: Bilderball **[Tag08]**

Eine umfassende Bestandsaufnahme über Visualisierungsmethoden für Dateien, sowohl generisch für alle Dateien (z.B. Kalenderansicht) als auch typspezifisch (z.B. Bilderball für visuelle Inhalte), ist zur Zeit Gegenstand einer an der Fachhochschule Dortmund durchgeführten und vom Autor betreuten Projektarbeit mit daran anschließender Bachelor-Thesis.

8.4 Ausblick

Die in dieser Arbeit vorgestellten Konzepte sind nicht nur für vollwertige Betriebssysteme geeignet, sondern auch für den Einsatz in Umgebungen, bei denen Eingaben durch den Benutzer nur eingeschränkt möglich sind. Neben mobilen Geräten, insbesondere MP3-Playern und Smartphones, stellt »surface computing« eine solche Plattform dar.

Bei »Microsoft Surface« **[Mic08b]** handelt es sich um einen Tisch mit berührungsempfindlicher Oberfläche, die mehrere Berührungspunkte gleichzeitig registrieren kann. Spezielle Applikationen sind für den Einsatz in Bars, Casinos, Geschäften und anderen Orten konzipiert. Gerade wegen der innovativen Bedienung dieses Computersystems sind die Eingabemöglichkeiten für den Benutzer sehr beschränkt: eine Maus oder Tastatur ist nicht vorhanden, und Menupunkte auf der Bildschirmoberfläche müssen großflächig genug sein, um durch eine Berührung sicher getroffen zu werden:

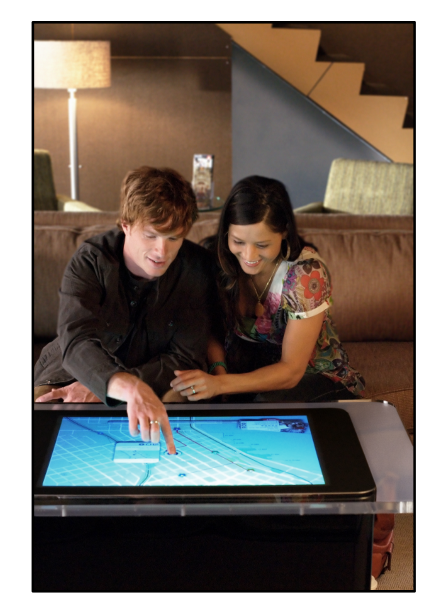


Abb. 8.4-1: Surface **[Mic08b]**



Abb. 8.4-2: Foto-Organisation mit Microsoft Surface **[Mic08b]**

Die nach Meinung des Autors herausragendsten Anwendungen für derartige Systeme ergeben sich im Zusammenspiel mit Geräten, die von der Oberfläche erkannt und über WLAN oder Bluetooth ins System eingebunden werden können (→ *Abb. 8.4-3*, → *Abb. 8.4-4*). Gerade hier bietet eine Library aufgrund der Vielzahl möglicher Dateiattribute große Vorteile gegenüber herkömmlichen Dateisystemen.

Als Beispiel sollen hier wiederum Mobiltelefone und MP3-Player angeführt werden. Wird ein solches Gerät vom Surface-System erkannt, wird ein Kreis um das Gerät dargestellt. Zusätzlich erscheinen einige der auf dem Gerät gespeicherten Bilder oder Musikstücke, letztere dargestellt durch eine Art Karteikarte mit Titelbild, Bewertung und Zusatzinformationen (→ *Abb. 8.4-4*). Der Benutzer kann nun über ein einfaches Menu Fotos versenden (etwa an einen Onlinedienst), oder beliebige Dateien mit einer Fingerbewegung in den Kreis eines anderen Geräts ziehen und dadurch übertragen. Dateien, die in einen zusätzlich dargestellten Bereich bewegt werden, speichert das jeweilige Surface-System dauerhaft ab:

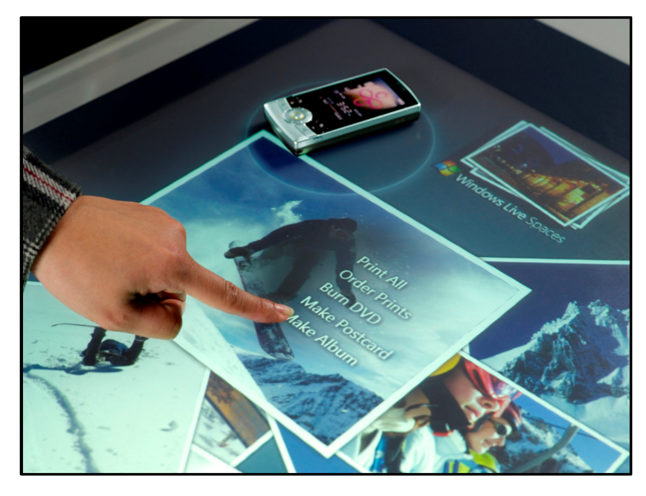


Abb. 8.4-3: Foto-Applikation **[Mic08b]**



Abb. 8.4-4: Interaktion mit Geräten **[Mic08b]**

In diesem Szenario kann das Library-Modell seine Stärken voll ausspielen. Ein grundlegender Vorteil ist die inhärente Typsicherheit für Dateien, so dass beim Datentransfer von und zu Geräten nur unterstützte Dateiformate, insbesondere für weniger standardisierte Anwendungen wie Navigationssysteme, übertragen werden. Hierzu wird abhängig vom erkannten Gerät für das dort eingebaute Dateisystem ein Library-Schema verwaltet oder übertragen. Der Zugriff auf beliebige Attribute und Metadaten unterstützt die Visualisierung von Datenobjekten wie Audio-Inhalten (→ *Abb. 8.4-4*), insbesondere da eine hohe Geschwindigkeit durch den Einsatz eines Master/Slave-Indexes gewährleistet werden kann. Dadurch treten durch das Anzeigen der enthaltenen Dateien keine spürbaren Verzögerungen beim Platzieren eines Geräts auf der Oberfläche auf.

Darüber hinaus gestatten es die flexiblen Objekt-Schemata, oberflächenspezifische Attribute zu einzelnen Dateien zu speichern, etwa Position und Zustand nach der letzten Benutzung. Dies vereinfacht die Bedienung, da Einstellungen (etwa die Größe und Lage eines Bildes) schnell und automatisch wiederhergestellt werden können.

Anhang A Glossar

BLOB	Abkürzung für »Binary Large Object«, ein Wertebereich für Attribute zur Speicherung von Dateien in relationalen Datenbanken
DBMS	Abkürzung für »Datenbank Management System«
ext2	Wichtiges Dateisystem für Linux, das auf *I-Nodes* basiert und Verzeichniseinträge in Bäumen abspeichert **[Kol07b]**
Domain	Speicherort (→ *Kap. 6.1.1*)
FAT	Mit DOS eingeführtes Dateisystem, das zur Kompatibilität für nahezu alle externen Flash-Speicherkarten und USB-Sticks verwendet wird; Datenblöcke einer Datei werden als verkettete Liste gespeichert, wodurch ein Dateizeiger in $O(n)$ bewegt werden kann **[Kol07b]**
Filter	Datenstuktur, die in der Referenz-Library (→ *Kap. 6*) die Parameter einer Suchanfrage enthält (→ *Kap. 6.2.2*)
Finder	*Shell* von Mac OS
I-Node	»Information Node«, enthält in vielen Unix-Dateisystemen wie z.B. *ext2* alle Informationen zu einer Datei, einschließlich der von ihr belegten Blöcke auf dem Datenträger; ein Verschieben des Dateizeigers ist in $O(1)$ möglich **[Kol07b]**
Leerlauf-Prozess	Prozess, der nur ausgeführt wird, wenn kein anderer Prozess CPU-Zeit beansprucht
Library	Datenspeicher, auf den sowohl relationale Datenbanken als auch Dateisysteme abgebildet werden können (→ *Kap. 4*)
Mountpoint	Spezielles Verzeichnis der Unix-Verzeichnishierarchie, das mit dem Hauptverzeichnis eines Dateisystems zusammenfällt; alle untergeordneten Verzeichnisse gehören nicht mehr zum Root-Dateisystem.
NTFS	Standard-Dateisystem von Microsoft Windows; kann für eine Datei mehrere Dateikörper (»Forks«) verwalten **[Kol07b]**

Query containment	Eingrenzen von Suchanfragen (→ *Kap. 5.2.2*, → *Kap. 6.2.2.1*)
Registry	Konfigurationsdatei von Microsoft Windows; wird heute auch zum Speichern von Programmeinstellungen benutzt (→ *Kap. 2.4.2*)
Semantisches Dateisystem	Ein semantisches Dateisystem arbeitet typabhängig, kann also individuelle Datei-Schemata verwalten (→ *Kap. 4.3.1*)
Shell	Anwendungsprogramm, das nach dem Booten oder nach dem Login als erstes gestartet wird und einen Mechanismus zur Eingabe oder Auswahl von Befehlen bietet und andere Programme starten kann (→ *Kap. 7.1*)
SQL	»Structured Query Language«, Abfragesprache für Datenbanken
Tagging	Versehen von Dateien mit Zusatzinformationen (→ *Kap. 7.5*)
Widget	Kleines Programm, das ohne Fenster direkt auf dem Desktop dargestellt wird und kleine Aufgaben, wie z.B. die Anzeige der Uhrzeit, übernimmt (→ *Kap. 7.1.1.1*)

Anhang B
Testprogramm zur Seek-Geschwindigkeit

Um in → *Kap. 3.3.3.1* den Geschwindigkeitsunterschied zwischen sequenziellem Lesen und dem Lesen von Sektoren an beliebigen Positionen zu ermitteln, wurde das folgende Testprogramm erstellt. Es kann mit Borland Pascal für den Real Mode compiliert werden.

Für beide Zugriffsarten werden jeweils 10 Durchläufe ausgeführt, bei jedem Durchlauf werden 1024 KB, also 2048 Sektoren, eingelesen. Für den sequenziellen Zugriff wird bei jedem Durchlauf ein zufälliger Startblock ausgewählt (also 10 Mal), beim randomisierten Zugriff für jeden einzelnen Sektor (also 20480 Mal). Anschließend wird pro Zugriffsart die Gesamtzeit für die 10 Durchläufe ausgegeben, mit einer Genauigkeit von einer achtzehntel Sekunde.

B.1 SEEKTEST.PAS

```
program SeekTest;
uses Dos;
```

Die folgende Funktion ruft das BIOS auf, um **Anz** Sektoren ab Sektor **LBA** an die durch **Buf** spezifizierte Adresse zu laden. Die entsprechende Funktion, **42h**, arbeitet direkt mit LBA-Adressen.

```
function Read(Anz: Byte; LBA: LongInt; Buf: Pointer): Boolean;
type
  ExtType=record
    Size,Anz: Word;
    P: Pointer;
    Adr,Dummy: LongInt;
end;
var
  R: Registers;
  Ext: ExtType;
  Cyl,Head,Sec: Word;
begin
  FillChar(Ext,SizeOf(Ext),0);
  Ext.Size:=$10;
  Ext.Anz:=Anz;
  Ext.P:=Buf;
  Ext.Adr:=LBA;
  R.AH:=$42;
  R.DL:=$80;
  R.DS:=Seg(Ext);
  R.SI:=Ofs(Ext);
  Intr($13,R);
  Read:=(R.Flags and fCarry)=0;
end;
```

Die Zeitmessung erfolgt durch Einklinken in den Zeit-Interrupt (IRQ 0, Interrupt 08h). Dieser IRQ wird genau $^{1193180}/_{65536}$, also etwa 18,2 Mal pro Sekunde, aufgerufen **[Tis94]**.

```
var
  OldInt8: Pointer;
  Clock: LongInt;

procedure NewInt8; assembler;
Asm
  PUSH AX
  PUSH DS
  MOV AX,SEG @DATA
  MOV DS,AX
  INC DWORD PTR DS:[Clock]
  PUSHF
  CALL DWORD PTR OldInt8
  POP DS
  POP AX
  IRET
end;

const MaxLBA=0;              {Größe der Festplatte abzüglich 2048 Sektoren hier eintragen !}
var
  P: Pointer;
  Start,LBA: LongInt;
  Durchlauf,Chunk: Word;
begin
  GetMem(P,32768);
  GetIntVec($08,OldInt8);
  SetIntVec($08,@NewInt8);
{Sequenziell}
  Clock:=0;
    for Durchlauf:=1 to 10 do begin
    write(#13+'Sequenziell ',Durchlauf);
    LBA:=Random(MaxLBA);
      for Chunk:=1 to 32 do begin
      Read(64,LBA,P);
      Inc(LBA,64);
      end;
    end;
  writeln(#13+'Sequenziell: ',Clock,' Ticks zu je 1/18.2s');
{Random}
  Clock:=0;
    for Durchlauf:=1 to 10 do begin
    write(#13+'Random ',Durchlauf);
      for Chunk:=1 to 2048 do begin
      LBA:=Random(MaxLBA);
      Read(1,LBA,P);
      end;
    end;
  writeln(#13+'Random: ',Clock,' Ticks zu je 1/18.2s');
  SetIntVec($08,OldInt8);
  FreeMem(P,32768);
end.
```

B.2 Messergebnisse

Die folgende Tabelle zeigt die Messergebnisse des Testprogramms (→ *Kap. B.1*) für verschiedene Datenträger. Der obere Tabellenabschnitt beschreibt mechanische Festplatten, die beiden untersten Zeilen SSDs (»Solid State Discs«, die auf Flash-Speicher basieren):

	Größe	Sequenziell	Zufällig	Faktor
Toshiba MK1924FCV	518 MB	8.330 ms	469.176 ms	56,3
Western Digital WDC AC21000H	1 GB	2.291 ms	339.670 ms	148,3
Fujitsu M1614TA	1 GB	2.407 ms	324.286 ms	134,7
Toshiba MK2001MPL	2 GB	2.582 ms	394.335 ms	152,7
Fujitsu MPC304AT	4 GB	2.720 ms	311.099 ms	114,4
Seagate ST310212A	10 GB	4.291 ms	365.659 ms	85,2
IBM DARA-21200	12 GB	3.040 ms	942.731 ms	310,1
Maxtor 31536H2	15 GB	2.813 ms	446.814 ms	158,8
ExcelStore ES3220	20 GB	4.516 ms	395.879 ms	87,6
Samsung SV2042H	20 GB	2.159 ms	202.802 ms	93,9
Seagate ST330630A	30 GB	4.736 ms	283.956 ms	60,0
Samsung SV4011N	40 GB	4.038 ms	276.154 ms	68,4
Samsung SV0802N	80 GB	4.114 ms	231.040 ms	56,2
Toshiba MK8032GSX	80 GB	824 ms	232.033 ms	281,6
Samsung SV2508N	250 GB	3.044 ms	252.198 ms	82,9
Samsung HD321KJ	320 GB	1.384 ms	211.082 ms	152,5
Sony Memorystick Pro	512 MB	1.971 ms	3.850 ms	2,0
Transcend 2,5" SSD	32 GB	604 ms	6.429 ms	10,6

Grün hervorgehoben: schnellster Datenträger
Rot hervorgehoben: langsamster Datenträger

Abb. B.2-1: Zeitverhalten von Speichermedien

Anhang C
Testprogramm »Microsoft SQL-Server 2005«

Um in → *Kap. 3.3.5.1* die Leistung des Microsoft SQL-Server zu messen, wurde ein Testprogramm in C# entwickelt. Das Programm (→ *Kap. C.3*) kann mit Microsoft Visual Studio für die .NET-Umgebung compiliert werden. Zur korrekten Ausführung ist neben einem installierten Microsoft SQL-Server die Bibliothek **SQLXML** erforderlich, die vor dem Compilieren des Programms in das Projekt eingebunden werden muss:

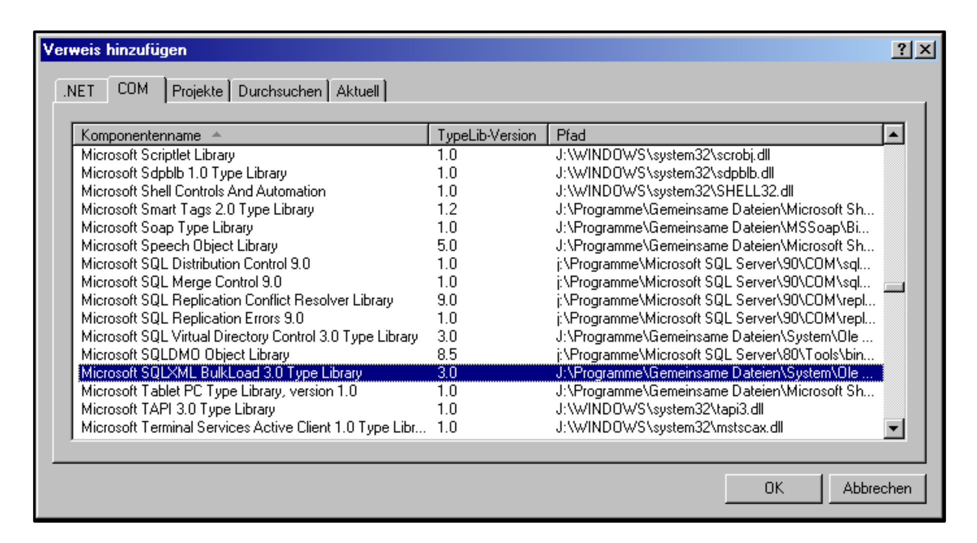


*Abb. C-1: Verweis auf die Bibliothek **SQLXML** in Microsoft Visual Studio*

C.1 METADATA.XML

Alle Attribute der Testdateien wurden in einer XML-Datei gespeichert. Dieses Dateiformat kann von vielen Applikationen (durch die Bibliothek **SQLXML** auch vom Microsoft SQL-Server) importiert werden.

```
<ROOT>
 <File>
  <FileTyp>34</FileTyp>
  <FileBez>Sabine 1</FileBez>
  <FileKey>(1P~L_8S</FileKey>
  <FileSize>632690</FileSize>
  <FileTime>18.07.2005, 18:03:58</FileTime>
  <FileNew>0</FileNew>
  <FileIATA>9313</FileIATA>
  <IMGL>1200</IMGL>
  <IMGH>1600</IMGH>
  <IMGFarbmodus>1</IMGFarbmodus>
  <IMGCreationTime>17:56</IMGCreationTime>
  <IMGCreationYear>2005</IMGCreationYear>
  <IMGCreationMonth>7</IMGCreationMonth>
  <IMGCreationDay>17</IMGCreationDay>
```

```
    <IMGEquipment>Minolta Co., Ltd. DiMAGE X20</IMGEquipment>
    <IMGBelichtung>1/180 Sekunde</IMGBelichtung>
    <IMGFirmware>Ver. 1.00</IMGFirmware>
    <IMGTitel>MINOLTA DIGITAL CAMERA</IMGTitel>
    <IMGFocus>4,80mm</IMGFocus>
    <IMGBlende>f/28</IMGBlende>
  </File>
  ...
</ROOT>
```

C.2 METADATA.XSD

Das folgende Schema wird verwendet, um die Metadaten aus der Data **METADATA.XML** (→ *C.1*) mit **SQLXML** in die Datenbank zu importieren.

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:xml:datatypes"
        xmlns:sql="urn:schemas-microsoft-com:xml-sql" >

  <ElementType name="FileTyp" dt:type="int" />
  <ElementType name="FileBez" dt:type="string" />
  <ElementType name="FileKey" dt:type="string" />
  <ElementType name="FileSize" dt:type="int" />
  <ElementType name="FileTime" dt:type="string" />
  <ElementType name="FileNew" dt:type="int" />
  <ElementType name="FilePassword" dt:type="string" />
  <ElementType name="FileIATA" dt:type="int" />
  ...

  <ElementType name="ROOT" sql:is-constant="1">
    <element type="File" />
  </ElementType>

  <ElementType name="File" sql:relation="Test">
    <element type="FileTyp" sql:field="FileTyp" />
    <element type="FileBez" sql:field="FileBez" />
    <element type="FileKey" sql:field="FileKey" />
    <element type="FileSize" sql:field="FileSize" />
    <element type="FileTime" sql:field="FileTime" />
    <element type="FileNew" sql:field="FileNew" />
    <element type="FilePassword" sql:field="FilePassword" />
    <element type="FileIATA" sql:field="FileIATA" />
    ...
  </ElementType>
</Schema>
```

C.3 PROGRAM.CS

Die Datei **PROGRAM.CS** enthält das eigentliche Testprogramm. Die Klasse **Program** greift auf eine nicht abgebildete Klasse **StopWatch** zu, deren Objekte die Zeit mit einer Genauigkeit von 0,1 ms messen können. Intern greift das .NET-Framework dabei auf den **RDTSC**-Befehl **[She96]** zurück.

```
using SQLXMLBULKLOADLib;
using System;
using System.Data;
using System.Data.Sql;
using System.Data.SqlClient;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
   class Program
   {
      //Nur ein Thread - SQLXML ist nicht thread-safe
      [STAThread]
      static void Main(string[] args)
      {
         SqlConnection conn = new System.Data.SqlClient.SqlConnection();
         conn.ConnectionString =
            "integrated security=SSPI;data source=CARMIN\\SQLEXPRESS;" +
            "persist security info=False;initial catalog=Metadata";
         Console.Write("Connecting... ");
         conn.Open();
         Console.WriteLine("done");

         //Alte Tabelle entfernen
         SqlCommand cmd = new SqlCommand("DROP TABLE Test", conn);
         try
         {
            cmd.ExecuteNonQuery();
         }
         catch
         {
         }

         //Neue Tabelle erstellen
         cmd = new SqlCommand("CREATE TABLE Test (" +
            "FileTyp int," +
            "FileBez varchar(31)," +
            "FileKey varchar(8)," +
            "FileSize int," +
            "FileTime varchar(20)," +
            "FileNew int," +
            "FilePassword varchar(15)," +
            "FileIATA int," +
            "VIDFourCC varchar(4)," +
            "VIDL int," +
            "VIDH int," +
            "VIDVideoDecoder int," +
            "VIDAudioDecoder int," +
            "VIDCreationYear int," +
            "VIDCreationMonth int," +
            "VIDCreationDay int," +
            "VIDCompressedHeader int," +
            "VIDMultiStreams int," +
            "VIDArtist varchar(31)," +
            "VIDComment varchar(31)," +
            "VIDThema varchar(31)," +
            "VIDVideoTitel varchar(31)," +
            "VIDFirmware varchar(31)," +
            "ID3Artist varchar(30)," +
            "ID3Trackname varchar(30)," +
            "ID3Albumname varchar(30)," +
```

```
        "ID3Comment varchar(30)," +
        "ID3Year varchar(4)," +
        "ID3Genre int," +
        "IMGL int," +
        "IMGH int," +
        "IMGFarbmodus int," +
        "IMGCreationTime varchar(5)," +
        "IMGCreationYear int," +
        "IMGCreationMonth int," +
        "IMGCreationDay int," +
        "IMGEquipment varchar(127)," +
        "IMGBelichtung varchar(127)," +
        "IMGFirmware varchar(127)," +
        "IMGTitel varchar(127)," +
        "IMGCopyright varchar(127)," +
        "IMGArtist varchar(127)," +
        "IMGComment varchar(127)," +
        "IMGFocus varchar(15)," +
        "IMGBlende varchar(15)," +
        "IMGChip varchar(11)," +
        "AUDMin int," +
        "AUDSec int," +
        "AUDHund int," +
        "AUDChannels int," +
        "AUDBits int," +
        "AUDSamples int," +
        "AUDSampleRate int," +
        "LSTAnz1 int," +
        "LSTAnz2 int," +
        "LSTAnz3 int," +
        "LSTAnz4 int," +
        "LSTOptionen int," +
        "LSTCreationTime varchar(5)," +
        "LSTCreationYear int," +
        "LSTCreationMonth int," +
        "LSTCreationDay int" +
        ")", conn);
    cmd.ExecuteNonQuery();

    //Indexe erstellen
    cmd = new SqlCommand("CREATE CLUSTERED INDEX fi ON Test (" +
        "FileTyp,FileBez,FileKey,FileSize,FileTime,FileNew,FilePassword," +
        "FileIATA,IMGL,IMGH,ID3Artist);", conn);
    cmd.ExecuteNonQuery();
    cmd = new SqlCommand("CREATE NONCLUSTERED INDEX vid ON Test (" +
        "VIDL,VIDH,VIDVideoDecoder,VIDAudioDecoder,VIDCreationYear," +
        "VIDCreationMonth,VIDCreationDay,VIDCompressedHeader,VIDMultiStreams," +
        "VIDArtist,VIDComment,VIDThema,VIDVideoTitel,VIDFirmware);", conn);
    cmd.ExecuteNonQuery();
    cmd = new SqlCommand("CREATE NONCLUSTERED INDEX id3 ON Test (" +
        "ID3Artist,ID3Trackname,ID3AlbumName,ID3Comment,ID3Year,ID3Genre);", conn);
    cmd.ExecuteNonQuery();
    cmd = new SqlCommand("CREATE NONCLUSTERED INDEX img ON Test (" +
        "IMGL,IMGH,IMGFarbmodus,IMGCreationTime,IMGCreationYear,IMGCreationMonth,"+
        "IMGCreationDay,IMGEquipment,IMGBelichtung,IMGFirmware,IMGTitel," +
        "IMGCopyright,IMGArtist,IMGComment,IMGFocus,IMGBlende);", conn);
    cmd.ExecuteNonQuery();
    cmd = new SqlCommand("CREATE NONCLUSTERED INDEX aud ON Test (" +
        "AUDMin,AUDSec,AUDHund,AUDChannels,AUDBits,AUDSamples,AUDSampleRate" +
        ");", conn);
    cmd.ExecuteNonQuery();
```

```
cmd = new SqlCommand("CREATE NONCLUSTERED INDEX lst ON Test (" +
   "LSTAnz1,LSTAnz2,LSTAnz3,LSTAnz4,LSTOptionen,LSTCreationTime," +
   "LSTCreationYear,LSTCreationMonth,LSTCreationDay);", conn);
cmd.ExecuteNonQuery();

StopWatch sw = new StopWatch();

//Metadaten laden
sw.Reset();
SQLXMLBulkLoad3Class objXBL = new SQLXMLBulkLoad3Class();
objXBL.ConnectionString = "Provider=sqloledb;server=CARMIN\\SQLEXPRESS;" +
   "database=Metadata;integrated security=SSPI";
objXBL.ErrorLogFile = "SQLXML3Books.errlog";
objXBL.KeepIdentity = false;
objXBL.Execute("J:\\Dokumente und Einstellungen\\ROOT\\" +
   "Eigene Dateien\\DISS\\SRC\\METADATA.XSD",
   "J:\\Dokumente und Einstellungen\\ROOT\\" +
   "Eigene Dateien\\DISS\\SRC\\METADATA.XML");
Console.WriteLine("Load: " + sw.Peek() / 10.0 + " ms");

//Testfall 1
sw.Reset();
cmd = new SqlCommand("SELECT * FROM Test WHERE (FileNew = 1)", conn);
SqlDataReader r = cmd.ExecuteReader();
while (r.Read())
{
}
r.Close();
Console.WriteLine("Case 1: " + sw.Peek() / 10.0 + " ms");

//Testfall 2
sw.Reset();
cmd = new SqlCommand("SELECT * FROM Test WHERE (FileTyp = 6 OR FileTyp = 15 " +
   "OR FileTyp = 26 OR FileTyp = 34)", conn);
r = cmd.ExecuteReader();
while (r.Read())
{
}
r.Close();
Console.WriteLine("Case 2: " + sw.Peek() / 10.0 + " ms");

//Testfall 3
sw.Reset();
cmd = new SqlCommand("SELECT * FROM Test WHERE (FileTyp = 5 OR FileTyp = 7 " +
   "OR FileTyp = 23 OR FileTyp = 32)", conn);
r = cmd.ExecuteReader();
while (r.Read())
{
}
r.Close();
Console.WriteLine("Case 3: " + sw.Peek() / 10.0 + " ms");

//Testfall 4
sw.Reset();
cmd = new SqlCommand("SELECT * FROM Test WHERE ((FileTyp = 6 OR FileTyp = 15 " +
   "OR FileTyp = 26 OR FileTyp = 34) AND IMGL >= 1024)", conn);
r = cmd.ExecuteReader();
while (r.Read())
{
}
r.Close();
Console.WriteLine("Case 4: " + sw.Peek() / 10.0 + " ms");
```

```
            //Testfall 5
            sw.Reset();
            cmd = new SqlCommand("SELECT * FROM Test WHERE ((FileTyp = 5 OR FileTyp = 7 " +
                "OR FileTyp = 23 OR FileTyp = 32) AND ID3Artist = 'Anastacia')", conn);
            r = cmd.ExecuteReader();
            while (r.Read())
            {
            }
            r.Close();
            Console.WriteLine("Case 5: " + sw.Peek() / 10.0 + " ms");

            conn.Close();
            Console.WriteLine("Ready !");
            Console.ReadLine();
        }
    }
}
```

Anhang D
Testprogramm »PostgreSQL«

Um in → *Kap. 3.3.5.2* die Leistung der Datenbank PostgreSQL zu messen, wurde ein Testprogramm in C# entwickelt. Das Programm (→ *Kap. D.1*) kann mit Microsoft Visual Studio für die .NET-Umgebung compiliert werden. Zur korrekten Ausführung ist neben einem installierten Microsoft SQL-Server ein passender ODBC-Treiber für PostgreSQL erforderlich.

D.1 PROGRAM.CS

Die Datei PROGRAM.CS enthält das eigentliche Testprogramm. Die Klasse Program greift auf eine nicht abgebildete Klasse StopWatch zu, deren Objekte die Zeit mit einer Genauigkeit von 0,1 ms messen können. Intern greift das .NET-Framework dabei auf den RDTSC-Befehl **[She96]** zurück.

```
using System;
using System.Data;
using System.Data.OleDb;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
   class Program
   {
      //Nur ein Thread - OleDb ist nicht thread-safe
      [STAThread]
      static void Main(string[] args)
      {
         Console.Write("Connecting... ");
         OleDbConnection conn = new OleDbConnection("Provider=PostgreSQL;" +
            "Data Source=localhost;password=E61534S;User ID=ROOT;location=Metadata;");
         conn.Open();
         Console.WriteLine("done");

         //Alte Tabelle entfernen
         OleDbCommand cmd = new OleDbCommand("DROP TABLE Test", conn);
         try
         {
            cmd.ExecuteNonQuery();
         }
         catch
         {
         }
```

```
//Neue Tabelle erstellen
cmd = new OleDbCommand("CREATE TABLE Test (" +
  "FileTyp int," +
  "FileBez varchar(31)," +
  "FileKey varchar(8)," +
  "FileSize int," +
  "FileTime varchar(20)," +
  "FileNew int," +
  "FilePassword varchar(15)," +
  "FileIATA int," +
  "VIDFourCC varchar(4)," +
  "VIDL int," +
  "VIDH int," +
  "VIDVideoDecoder int," +
  "VIDAudioDecoder int," +
  "VIDCreationYear int," +
  "VIDCreationMonth int," +
  "VIDCreationDay int," +
  "VIDCompressedHeader int," +
  "VIDMultiStreams int," +
  "VIDArtist varchar(31)," +
  "VIDComment varchar(31)," +
  "VIDThema varchar(31)," +
  "VIDVideoTitel varchar(31)," +
  "VIDFirmware varchar(31)," +
  "ID3Artist varchar(30)," +
  "ID3Trackname varchar(30)," +
  "ID3Albumname varchar(30)," +
  "ID3Comment varchar(30)," +
  "ID3Year varchar(4)," +
  "ID3Genre int," +
  "IMGL int," +
  "IMGH int," +
  "IMGFarbmodus int," +
  "IMGCreationTime varchar(5)," +
  "IMGCreationYear int," +
  "IMGCreationMonth int," +
  "IMGCreationDay int," +
  "IMGEquipment varchar(127)," +
  "IMGBelichtung varchar(127)," +
  "IMGFirmware varchar(127)," +
  "IMGTitel varchar(127)," +
  "IMGCopyright varchar(127)," +
  "IMGArtist varchar(127)," +
  "IMGComment varchar(127)," +
  "IMGFocus varchar(15)," +
  "IMGBlende varchar(15)," +
  "IMGChip varchar(11)," +
  "AUDMin int," +
  "AUDSec int," +
  "AUDHund int," +
  "AUDChannels int," +
  "AUDBits int," +
  "AUDSamples int," +
  "AUDSampleRate int," +
  "LSTAnz1 int," +
  "LSTAnz2 int," +
  "LSTAnz3 int," +
  "LSTAnz4 int," +
  "LSTOptionen int," +
  "LSTCreationTime varchar(5)," +
  "LSTCreationYear int," +
```

```
    "LSTCreationMonth int," +
    "LSTCreationDay int" +
    ")", conn);
cmd.ExecuteNonQuery();

StopWatch sw = new StopWatch();

//Metadaten laden
DataSet ds = new DataSet();
ds.ReadXml("J:\\Dokumente und Einstellungen\\ROOT\\" +
    "Eigene Dateien\\DISS\\SRC\\METADATA.XML");

String prefix = "INSERT INTO Test (";
int cols = ds.Tables[0].Columns.Count;

//Indexe erstellen
for (int a = 0; a < cols; a++)
{
    cmd = new OleDbCommand("CREATE INDEX idx"+a.ToString()+" ON Test USING btree ("
        +ds.Tables[0].Columns[a].ToString()+")",conn);
    cmd.ExecuteNonQuery();
}

for (int a = 0; a < cols; a++)
{
    prefix = prefix + ds.Tables[0].Columns[a].ToString();
    if (a < cols - 1)
    {
        prefix = prefix + ",";
    }
}

sw.Reset();
foreach (DataRow f in ds.Tables[0].Rows)
{
    String st = prefix + ") VALUES (";
    for (int a = 0; a < cols; a++)
    {
        String v = f[a].ToString().Replace("'", "_");
        if (v == "")
        {
            v = "0";
        }
        st = st + "'" + v + "'";
        if (a < cols - 1)
        {
            st = st + ",";
        }
    }
    st = st + ")";
    cmd = new OleDbCommand(st, conn);
    cmd.ExecuteNonQuery();
}
Console.WriteLine("Load: " + sw.Peek() / 10.0 + " ms");
```

```
            //Testfall 1
            sw.Reset();
            cmd = new OleDbCommand("SELECT * FROM Test WHERE (FileNew = 1)", conn);
            OleDbDataReader r = cmd.ExecuteReader();
            while (r.Read())
            {
            }
            r.Close();
            Console.WriteLine("Case 1: " + sw.Peek() / 10.0 + " ms");

            //Testfall 2
            sw.Reset();
            cmd = new OleDbCommand("SELECT * FROM Test WHERE (FileTyp = 6 OR FileTyp = 15 "+
                "OR FileTyp = 26 OR FileTyp = 34)", conn);
            r = cmd.ExecuteReader();
            while (r.Read())
            {
            }
            r.Close();
            Console.WriteLine("Case 2: " + sw.Peek() / 10.0 + " ms");

            //Testfall 3
            sw.Reset();
            cmd = new OleDbCommand("SELECT * FROM Test WHERE (FileTyp = 5 OR FileTyp = 7 "+
                "OR FileTyp = 23 OR FileTyp = 32)", conn);
            r = cmd.ExecuteReader();
            while (r.Read())
            {
            }
            r.Close();
            Console.WriteLine("Case 3: " + sw.Peek() / 10.0 + " ms");

            //Testfall 4
            sw.Reset();
            cmd = new OleDbCommand("SELECT * FROM Test WHERE ((FileTyp = 6 "+
                "OR FileTyp = 15 OR FileTyp = 26 OR FileTyp = 34) AND IMGL >= 1024)", conn);
            r = cmd.ExecuteReader();
            while (r.Read())
            {
            }
            r.Close();
            Console.WriteLine("Case 4: " + sw.Peek() / 10.0 + " ms");

            //Testfall 5
            sw.Reset();
            cmd = new OleDbCommand("SELECT * FROM Test WHERE ((FileTyp = 5 OR FileTyp = 7 "+
                "OR FileTyp = 23 OR FileTyp = 32) AND ID3Artist = 'Anastacia')", conn);
            r = cmd.ExecuteReader();
            while (r.Read())
            {
            }
            r.Close();
            Console.WriteLine("Case 5: " + sw.Peek() / 10.0 + " ms");

            conn.Close();
            Console.WriteLine("Ready !");
            Console.ReadLine();
        }
    }
}
```

Literaturverweise

[Apa07] Apache Software Foundation
HTTPD
http://httpd.apache.org/
Stand: 23.10.2008 (erste Referenzierung: 22.10.2007)

[App05] Apple Corporation
Technology Brief, Mac OS X: Spotlight
http://images.apple.com/macosx/pdf/MacOSX_Spotlight_TB.pdf
Stand: 23.10.2008 (erste Referenzierung: 05.09.2005)

[App08] Apple Corporation
Mac OS X – Spotlight Plugins
http://www.apple.com/downloads/macosx/spotlight/
Stand: 23.10.2008

[Ben75] Bentley, J. L.
Multidimensional binary search trees used for associative searching
Communications of the ACM, Volume 18, Ausgabe 9, 1975
http://portal.acm.org/citation.cfm?id=361007
Stand: 23.10.2008

[Ber96] Berchtold, S. et al.
The X-tree: An Index Structure for High-Dimensional Data
Proceedings of the 22nd VLDB Conference, Mumbai 1996
http://www.vldb.org/conf/1996/P028.PDF
Stand: 23.10.2008

[Bor99] Borodin, A. et al.
Lower Bounds for High Dimensional Nearest Neighbor Search and Related Problems
Proceedings of the 31st ACM Symposium on Theory of computing, Atlanta 1999
http://portal.acm.org/citation.cfm?id=301330
Stand: 23.10.2008

[Bri03] Brinkhoff, T.
Hash-Bäume und andere mehrdimensionale Punktstrukturen zur Speicherung von Laserscanner-Daten
2. Oldenburger 3D-Tage
Optische 3D-Messtechnik – Photogrammetrie – Laser-Scanning, Wichmann-Verlag, 2003
http://www.fh-oow.de/institute/iapg/personen/brinkhoff/paper/3D-2003.pdf
Stand: 23.10.2008

[Cod70] Codd, E.
A Relational Model of Data for Large Shared Data Banks
Communications of the ACM, Volume 13, Ausgabe 6, Juni 1970
http://portal.acm.org/citation.cfm?id=362685
Stand: 23.10.2008

[Dou00] Dourish, P. et al.
Extending document management systems with user-specific active properties
ACM Transactions on Information Systems (TOIS), Volume 18 , Issue 2, April 2000
http://portal.acm.org/citation.cfm?id=348758
Stand: 23.10.2008

[EXI07] http://www.exif.org/
Stand: 23.10.2008 (erste Referenzierung: 03.12.2007)

[Fli07] FlickR
http://www.flickr.com/
Stand: 23.10.2008 (erste Referenzierung: 04.12.2007)

[Gia99] Giampaolo, D.
Practical File System Design with the Be File System
Morgan Kaufmann Publishers, 1. Auflage 1999

[Gif91] Gifford, D.K. et al.
Semantic file systems
Proceedings of the 13th ACM Symposium on Operating Systems Principles, Denver 1991
http://portal.acm.org/citation.cfm?id=121138
Stand: 23.10.2008

[Gno06] Gnome: the free desktop project
http://www.gnome.org/
Stand: 23.10.2008 (erste Referenzierung: 10.09.2006)

[Gop99] Gopal, B. et al.
Integrating Content-based Access Mechanisms with Hierarchical File Systems
Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, New Orleans 1999
http://www.usenix.org/events/osdi99/full_papers/gopal/gopal.pdf
Stand: 23.10.2008

[Gor04] Gorter, O.
Database File System – An Alternative to Hierarchy Based File Systems
Master's thesis, University of Twente, 2004
http://dbfs.sourceforge.net/
Stand: 23.10.2008 (erste Referenzierung: 05.09.2005)

[Gro07] Grobe, T. et al.
Geodaten für Fotos
Magazin für Computertechnik (c't), Special »Digitale Fotografie«
Heise-Verlag, 2007

[Gut84] Guttman, A
R-Trees: A Dynamic Index Structure for Spatial Searching
Proceedings of the ACM SIGMOD Conference, Boston 1984
http://www.sai.msu.su/~megera/postgres/gist/papers/gutman-rtree.pdf
Stand: 23.10.2008

[Hac99] Hacker, S. et al.
The BeOS Bible
Addison-Wesley-Verlag, 1. Auflage 1999
http://www.birdhouse.org/beos/bible/exc_data.html
Stand: 23.10.2008

[Har05] HardwareEcke.de
BeOS
http://www.hardwareecke.de/berichte/grundlagen/beos.php
Stand: 23.10.2008 (erste Referenzierung: 04.09.2005)

[IAT07] International Air Transport Association
http://www.iata.org/
Stand: 23.10.2008 (erste Referenzierung: 11.06.2007)

[Imf02] Imfeld, A.
Semantisches Filesystem ObjectSFS
Diplomarbeit, Eidgenössische Technische Hochschule Zürich
http://e-collection.ethbib.ethz.ch/ecol-pool/dipl/dipl_79.pdf
Stand: 23.10.2008

[Jag05] Jagadish, H.V. et al.
iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search
ACM Transactions on Database Systems (TODS), Volume 30, Issue 2, Juni 2005
http://portal.acm.org/citation.cfm?id=1071612
Stand: 23.10.2008

[KDE06] K Desktop Environment
http://www.kde.org/
Stand: 23.10.2008 (erste Referenzierung: 10.09.2006)

[Ker03] Kersten, M. et al.
A Database Striptease or How to Manage Your Personal Databases
Proceedings of the 29th VLDB Conference, Berlin 2003
http://www.vldb.org/conf/2003/papers/S34P01.pdf
Stand: 23.10.2008

[Kol03] Koll, K.
Analyse und Bewertung verschiedener wichtiger Videoformate
Technische Universität Dortmund, 12.07.2003
http://www.deskwork.de/DOWNLOAD/DOCS/DIPL-ARB.PDF
Stand: 23.10.2008

[Kol07a] Koll, K.
Master/slave index in computer systems
United States Patent 11/892071
http://www.freepatentsonline.com/y2008/0071732.html
Stand: 23.10.2008

[Kol07b] Koll, K.
Einführung in relationale Datenbanken und semantische Dateisysteme
http://www.deskwork.de/DOWNLOAD/DOCS/INTRO.PDF
Stand: 23.10.2008 (erste Referenzierung: 02.12.2007)

[Kol08a] Koll, K.
File systems should be like burger meals: supersize your allocation units !
Outrageous opinion statement, 6th USENIX Conference on File And Storage Technology, San Jose 2008
http://www.usenix.org/event/fast08/wips_posters/koll-wip.pdf
http://www.deskwork.de/DOWNLOAD/DOCS/FAST08.ZIP
Stand: 23.10.2008

[Kol08b] Koll, K.
A relational file system as an example for tailor-made DMS
Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management, Nantes 2008
http://portal.acm.org/citation.cfm?id=1385489
Stand: 23.10.2008

[Kol08c] Koll, K.
Indexing file attributes with the master/slave index
Postgraduate Symposium of the 2008 annual research conference of the South African institute of computer scientists and information technologists on Riding the wave of technology, George (Wilderness) 2008
http://www.deskwork.de/DOWNLOAD/DOCS/SAICSIT8.PDF
Stand: 23.10.2008

[Kol08d] Koll, K.
Fancy fancy indexing
ApacheCon US 2008, New Orleans 2008
http://www.deskwork.de/DOWNLOAD/DOCS/APACHE08.ZIP
Stand: 23.10.2008

[Kre07] Kremp, M.
Tagging im Trend – Gemeinsam besser finden
Spiegel Online
http://www.spiegel.de/netzwelt/web/0,1518,464902,00.html
Stand: 23.10.2008

[Les06] Leser, U. et al.
Informationsintegration
DPunkt-Verlag, 1. Auflage 2006

[LHN03] Von FRA über GRU nach GIG
Lufthansa Newslink, Ausgabe 24, Oktober 2003
http://konzern.lufthansa.com/de/html/presse/newslink/archiv_2003/newslink_2003_10/index.html
Stand: 23.10.2008

[Lin05] linuxlog.de
Beagle Search Tool
http://linuxlog-archiv.usr-local-bin.de/beaglesearchtool.html
Stand: 23.10.2008 (erste Referenzierung: 07.09.2005)

[Luc06] Lucene
Index File Formats
http://lucene.apache.org/java/1_9_0/fileformats.html
Stand: 23.10.2008 (erste Referenzierung: 12.07.2006)

[Luc07] Lucene
PoweredBy
http://wiki.apache.org/jakarta-lucene/PoweredBy
Stand: 23.10.2008 (erste Referenzierung: 03.12.2007)

[Mal83] Malone, T. W.
How Do People Organize Their Desks? Implications for the Design of Office Infomation Systems
ACM Transactions on Office Information Systems, Vol. 1, No. 1, Januar 1983
http://portal.acm.org/citation.cfm?id=357430
Stand: 23.10.2008

[Mar03] Marsden, G.
Improving the usability of the hierarchical file system
ACM International Conference Proceeding Series; Vol. 47: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology, Fourways 2003
http://portal.acm.org/citation.cfm?id=954027
Stand: 23.10.2008

[Mic93] Microsoft Corporation
Verknüpfen und Einbetten von Objekten
Benutzerhandbuch Microsoft Windows for Workgroups, 1993

[Mic00] Microsoft Corporation
How to disable the Find Fast indexer
http://support.microsoft.com/kb/158705/EN-US/
Stand: 23.10.2008

[Mic05] Microsoft Corporation
Microsoft WinFS Beta 1 Documentation

[Mic07] Microsoft Corporation
SQL Server 2008 – Microsoft Data Platform Vision
http://www.idea-lab.biz/Resources/SQL-Server-2008.pdf
Stand: 23.10.2008 (erste Referenzierung: 09.12.2007)

[Mic08a] Microsoft Corporation
An introduction to »WinFS« OPath
http://msdn.microsoft.com/en-us/library/aa480689.aspx
Stand: 23.10.2008 (erste Referenzierung: 27.07.2008)

[Mic08b] Microsoft Corporation
Microsoft Surface
http://www.microsoft.com/surface/
Stand: 03.12.2008 (erste Referenzierung: 03.12.2008)

[Mic08c] Microsoft Corporation
Create Namespace Extensions for Windows Explorer with the .NET Framework
http://msdn.microsoft.com/en-us/magazine/cc188741.aspx
Stand: 04.12.2008 (erste Referenzierung: 04.12.2008)

[Mic08d] Microsoft Corporation
Understanding XML Schema
http://msdn.microsoft.com/en-us/library/aa468557.aspx
Stand: 11.12.2008 (erste Referenzierung: 11.12.2008)

[Mil00] Millstein, T.
Query Containment for Data Integration Systems
Proceedings of ACM Symposium on Principles of Database Systems (PODS), Dallas 2000
http://www.cs.ucla.edu/~todd/research/pods00.pdf
Stand: 23.10.2008

[Mil05] Mills, B.
Metadata Driven Filesystem
http://bryanmills.net/uploads/metafs/bmills-final.pdf
Stand: 23.10.2008 (erste Referenzierung: 25.11.2005)

[Nic06] Nickell, S.
A Cognitive Defense of Associative Interfaces for Object Reference
http://www.gnome.org/~seth/storage/associative-interfaces.pdf
Stand: 23.10.2008 (erste Referenzierung: 14.02.2006)

[Nil05] Nillson, M.
http://www.id3.org/
Stand: 23.10.2008 (erste Referenzierung: 14.07.2005)

[Nov05] Novell
SUSE Linux Professional 93
http://www.novell.com/de-de/products/linuxprofessional/beagle.html
Stand: 23.10.2008 (erste Referenzierung: 07.09.2005)

[OSC06] Object Services and Consulting Inc.
Semantic File Systems
http://www.objs.com/survey/OFSExt.htm
Stand: 23.10.2008 (erste Referenzierung: 14.01.2006)

[Reg02] The Register
Windows on a database – sliced and diced by BeOS vets
http://www.theregister.co.uk/2002/03/29/windows_on_a_database_sliced/
Stand: 23.10.2008

[RFC1094] RFC 1094
NFS: Network File System Protocol Specification
http://www.ietf.org/rfc/rfc1094.txt
Stand: 23.10.2008

[RFC2045] RFC 2045
Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies
http://www.ietf.org/rfc/rfc2045.txt
Stand: 23.10.2008

[RFC2425] RFC 2425
A MIME Content-Type for Directory Information
http://www.ietf.org/rfc/rfc2425.txt
Stand: 23.10.2008

[RFC2445] RFC 2445
Internet Calendaring and Scheduling Core Object Specification (iCalendar)
http://www.ietf.org/rfc/rfc2445.txt
Stand: 23.10.2008

[RFC3986] RFC3986
Uniform Resource Identifier (URI): Generic Syntax
http://www.ietf.org/rfc/rfc3986.txt
Stand: 23.10.2008

[RFC4122] RFC 4122
A Universally Unique Identifier (UUID) URN Namespace
http://www.ietf.org/rfc/rfc4122.txt
Stand: 23.10.2008

[Rob99] Robbins, A.
UNIX in a nutshell
O'Reilly-Verlag, 3. Ausgabe 1999

[Saa05] Saake, G. et al.
Datenbanken: Implementierungstechniken
mitp-Verlag, 2. Auflage 2005

[Sak00] Sakurai, Y. et al.
The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation
Proceedings of the 26nd VLDB Conference, Kairo 2000
http://www.vldb.org/conf/2000/P516.pdf
Stand: 23.10.2008

[Sel87] Sellis, T.K. et al.
The R+-Tree: A Dynamic Index for Multi-Dimensional Objects
Proceedings of the 13th VLDB Conference, Brighton 1987

[She96] Shemitz, J.
Using RDTSC for benchmarking
Visual Developer Magazine, Juni 1996
http://www.midnightbeach.com/rdtsc.html
Stand: 23.10.2008

[Sho93] Shoens, K. et al.
The Rufus System: Information Organization for Semi-Structured Data
Proceedings of the 19th VLDB Conference, Dublin 1993
http://www.vldb.org/conf/1993/P097.PDF
Stand: 23.10.2008

[Stu07] StudiVZ
http://www.studivz.net/
Stand: 23.10.2008 (erste Referenzierung: 09.07.2007)

[Tag08] Tag Galaxy
http://www.taggalaxy.de/
Stand: 03.12.2008 (erste Referenzierung: 03.12.2008)

[Tan02] Tanenbaum, A.S.
Moderne Betriebssysteme
Pearson Education, 2. Auflage 2002

[Web98] Weber, R. et al.
A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces
Proceedings of the 24nd VLDB Conference, New York 1998
http://www.vldb.org/conf/1998/p194.pdf
Stand: 23.10.2008

[Whi96] White, D.A. et al.
Similarity Indexing with the SS-tree
Proceedings of the 12th IEEE International Conference on Data Engineering, 1996
http://portal.acm.org/citation.cfm?id=655573
Stand: 29.01.2009

[Wik05] Wikipedia
WinFS
http://en.wikipedia.org/wiki/WinFS
Stand: 23.10.2008 (erste Referenzierung: 29.06.2005)

[Wik06] Wikipedia
Lucene
http://de.wikipedia.org/wiki/Lucene
Stand: 23.10.2008 (erste Referenzierung: 12.07.2006)

[Zan82] Zaniolo, C. et al.
Database relations with null values
Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Los Angeles 1982
http://portal.acm.org/citation.cfm?id=588117
Stand: 13.11.2008